TURING 图灵程序设计丛书

PACKT PUBLISHING

Introduction to JVM Languages

Java虚拟机基础教程

【荷】Vincent van der Leun 著

袁国忠 译

- 运用大量示例探讨Java、Scala、Clojure、Kotlin和Groovy的核心概念


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

作者简介

Vincent van der Leun

全栈工程师，Oracle数据库认证专家。8岁开始编程，熟悉多种语言和平台，维护着JVM Fanboy博客。目前就职于致力于现代电子商务解决方案的CloudSuite公司。

封面设计 | 83720326@qq.com
自 在 书 装 设 计



图灵程序设计丛书

Introduction to JVM Languages

Java虚拟机基础教程

【荷】Vincent van der Leun 著
袁国忠 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Java虚拟机基础教程 / （荷）文森特·范德利昂著 ；
袁国忠译. -- 北京 : 人民邮电出版社, 2018.2
（图灵程序设计丛书）
ISBN 978-7-115-47779-8

Ⅰ. ①J… Ⅱ. ①文… ②袁… Ⅲ. ①JAVA语言—程序
设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2018)第010917号

内 容 提 要

本书概述Java虚拟机（JVM）及其特性，并用大量示例详细介绍了Java、Scala、Clojure、Kotlin和Groovy这5种基于JVM的语言。具体而言，首先概述了Java平台，紧接着详细阐述了JVM，然后分别介绍了上述各种语言的基础知识和核心概念，并运用它们开发项目、创建应用程序。

本书适合所有Java开发人员以及对JVM感兴趣的读者。

◆ 著　　[荷] Vincent van der Leun
译　　袁国忠
责任编辑　岳新欣
执行编辑　李　敏
责任印制　周昇亮
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷
◆ 开本：800×1000　1/16
印张：19.5
字数：461千字　　2018年2月第1版
印数：1－3 500册　　2018年2月北京第1次印刷
著作权合同登记号　图字：01-2017-9187号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

前　　言

Java虚拟机（Java Virtual Machine，JVM）是一个成熟的全能型软件运行平台，可充分利用现代硬件的功能。虽然基于Java的应用程序一度被认为速度缓慢、体态臃肿且极耗内存，但多年后的今天，情况已得到极大的改善。基于云的主流服务和网站通常要同时为数以万计的用户提供服务，它们很多都使用了基于JVM的后端，这绝非偶然。

开发运行在JVM上的应用程序时，用得最多的语言无疑是Java，但其他语言也越来越流行。本书介绍5种基于JVM的语言：Java、Scala、Clojure、Kotlin和Groovy。在这些语言中，有静态类型的，也有动态类型的；有面向对象的编程语言，也有函数式编程语言。JVM多才多艺，能够支持所有这些类型的语言。

通过在一本书中介绍这些语言，让你能够通过创建示例项目来轻松地比较它们，从而有望找出你最喜欢的语言。

涵盖的内容

第1章简要地概述Java平台和Java虚拟机（JVM）。该章描述运行在JVM上的应用程序的常见用途，包括Web应用程序、大数据分析和物联网（Internet of Things，IoT），还介绍最重要的JVM概念，如即时编译器、类型系统和垃圾收集器。

第2章从技术角度更详细地阐述JVM，包括如何在主要的操作系统（Windows、macOS和Linux）上安装Java开发包（Java Development Kit，JDK）、JDK的组织结构、Java类库的组织结构，以及如何通过设置类路径（ClassPath）来运行基于JVM的应用程序。

第3章介绍Java基础知识，包括创建类以及根据它实例化对象、在类中添加方法和属性，以及Java访问限定符和其他限定符。另外，还讨论了其他一些概念，包括抽象类、接口、数组、集合和异常。最后，介绍了线程和lambda等高级概念。

第4章详细介绍如何使用Java语言创建简单的Web服务。在创建简单Web服务的过程中，使用的工具包括Eclipse IDE、构建工具Gradle、SparkJava（一个微型Web服务框架）等编程库，以及单元测试框架JUnit。

第5章讨论既是面向对象编程语言又是函数式编程语言的Scala。该章介绍了Scala的安装过程以及它自带的交互式shell的用法；通过使用这个交互式shell，你可动态地输入并执行Scala代码，而无需先对代码进行编译。另外，还讨论了Scala的面向对象功能和函数式编程功能。

第6章详细介绍如何使用流行工具包Akka创建一个基于控制台的简单应用程序。Akka是一个专门为编写可伸缩的应用程序而设计的工具包，这种应用程序能够充分利用现代的多核处理器。该章详细讨论了很多Akka概念，如基于Actor的系统。为构建项目，使用了Scala Build Tool（Scala Build Tool，SBT），还使用了ScalaTest库来编写单元测试。

第7章介绍Clojure的基础知识。Clojure是一种动态的函数式编程语言，其设计灵感来自并非面向对象的语言Lisp。与Scala一样，Clojure也自带了一个交互式shell，可用于执行该章提供的各个示例。该章还讨论了一种用于在多线程应用程序中处理状态的技术——代理。

第8章详细介绍如何开发两个较小的项目。其中一个项目基于函数式编程语言（尤其是Lisp）中常用的技术monad，另一个项目是一个Web应用程序，它是使用流行的微型Web框架Luminus开发的。构建这两个项目时，使用的构建工具都是Leiningen。

第9章讨论JetBrain推出的静态类型编程语言Kotlin。该章阐述了Kotlin提供的可安全地处理null的类型系统，讨论了数据类、lambda和内联函数等功能，还介绍了Kotlin的过程性编程功能。

第10章详细介绍如何使用工具包JavaFX创建一个基于GUI的桌面应用程序。为构建这个项目，使用了Apache Maven；而为查找并修复bug，使用了Eclipse IDE的调试器。

第11章介绍动态编程语言Groovy。Groovy是最先推出的JVM语言之一，虽然在很大程度上它是动态语言，但也支持编译静态类型的代码，该章对这两种使用方式都做了介绍。另外，该章还探索了Groovy开发包，这是随Groovy语言一起发布的一个库，包含大量的内置类。

第12章详细介绍如何使用Groovy创建一个Web服务。这个Web服务是使用Vert.x框架创建的，它使用Java Database Connectivity（JDBC）标准从内嵌的数据库管理系统获取数据，并使用Groovy开发包中的类来生成XML。

附录A介绍了另外5种基于JVM的语言，它们大多是主流语言的方言：Oracle Nashorn（JavaScript）、Jython（Python）、JRuby（Ruby）、Frege（Haskell）和Ceylon（Red Hat推出的一种静态类型语言）。

附录B给出了各章末尾的小测验的答案。

需要什么

为最大限度地发挥本书的作用，需要一台现代的笔记本电脑或台式机，它使用最新版的Windows、macOS或Linux（最好是Ubuntu）操作系统，且至少有4GB内存（但越大越好）。

本书假定读者对使用的操作系统有一定的了解，能够熟练地安装程序以及将目录添加到环境变量中。

为谁而写

本书是为对JVM感兴趣并想深入了解最流行的JVM开发语言的程序员编写的，并假定读者使用过支持面向对象编程的现代语言，如JavaScript、Python、C#、VB.NET或C++。

排版约定

为将不同类型的信息区分开来，本书使用了很多文本样式。下面列出其中一些样式及其含义。

正文中的代码、数据库表名、用户输入，使用如下样式："然后，我们对这个对象实例调用`setName`方法。"

代码块使用如下样式：

```
Product p = new Product();
p.setName("Box of biscuits");
```

对于代码块中要突出的部分，使用粗体：

```
public String getName() {
  return name;
}
```

命令行输入或输出使用如下样式：

```
nano /etc/profile
```

新术语和重要词语使用黑体。



此图标表示警告或重要的注意事项。



此图标表示提示和技巧。

读者反馈

欢迎提供反馈，请将你对本书的看法告诉我们：哪些方面是你喜欢的，哪些方面你不喜欢。读者的反馈对我们来说很重要，因为这可帮助我们推出可最大限度发挥其功效的著作。

要给我们提供反馈，只需向feedback@packtpub.com发送电子邮件，并在主题中指出书名。

如果你有擅长的主题，并有志于写书或撰稿，请参阅www.packtpub.com/authors的撰稿指南。

客户支持

购买本社出版的图书后，你将获得各种帮助，让你购买的图书最大限度地发挥其效用。

下载示例代码

你可使用自己的账户从http://www.packtpub.com下载本书的示例代码文件。如果你是从其他地方购买的本书，可访问http://www.packtpub.com/support并注册，以便我们通过电子邮件将示例代码文件发送给你。

要下载代码文件，请执行如下步骤。

(1) 登录本社网站或使用你的邮件地址注册。

(2) 将鼠标指向网页顶端的SUPPORT选项卡。

(3) 单击Code Downloads & Errata。

(4) 在Search框中输入书名。

(5) 选择要下载哪本书的示例代码文件。

(6) 从下拉列表中选择你是从哪里购买的。

(7) 单击Code Download。

下载文件后，使用如下软件的最新版将其解压缩：

- Windows系统请使用WinRAR或7-Zip
- Mac系统请使用Zipeg、iZip或UnRarX
- Linux系统请使用7-Zip或PeaZip

本书的示例代码也可从GitHub（https://github.com/PacktPublishing/Introduction-to-JVM-Languages）下载；我们提供了众多图书和视频的配套代码，你可从https://github.com/PacktPublishing下载。

下载彩色图片

我们还提供了一个PDF文件，其中包含了本书用到的屏幕截图和图表的彩色图片。这些彩色图片将有助于你更好地理解输出的变化，你可从http://www.ituring.com.cn/book/1990下载。

勘误

我们万分小心，力图让图书的内容准确无误，但即便如此，错误也在所难免。如果你在本社出版的图书中发现错误（无论是正文还是代码中的错误），请告诉我们，我们将感激不尽。通过这样做，你将让其他读者免遭同样的挫折，还可帮助我们改进该书的后续版本。无论你发现什么错误，都请告诉我们；为此你可访问http://www.packtpub.com/submit-errata，输入书名，单击链接Errata Submission Form，再输入你发现的错误的详情。[①]你提交的勘误得到确认后，将被上传到我们的网站或添加到既有的勘误列表中。

要查看已提交的勘误，请访问https://www.packtpub.com/books/content/support，并在搜索框中输入书名，Errata栏将列出你搜索的信息。

打击盗版

在网上发布盗版材料是个屡禁不绝的问题。在保护版权和许可方面，本社的态度非常严肃，如果你在网上看到本社作品的非法复制品，请马上把网址或网站名告诉我们，以便我们能够采取补救措施。

请通过copyright@packtpub.com与我们取得联系，并提供你怀疑的盗版材料的链接。

对于你为保护我们的作者和提供有价值内容的能力提供的帮助，我们感激不尽。

问题

无论你有什么与本书相关的问题，都可通过questions@packtpub.com与我们联系，我们将竭尽全力去解决。

电子书

扫描如下二维码，即可购买本书电子版。



① 中文版勘误可到www.ituring.com.cn/book/1990查看和提交。——编者注

致　　谢

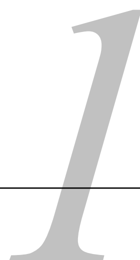
感谢Packt出版社的每位员工，是他们的辛勤劳动才让本书得以付梓。感谢编辑Nitin、Vikas和Subhalaxmi以及技术审校Ramasubramanian。感谢父亲Anton和母亲Irene以及兄弟Alexander、Ruben和Wendy的大力支持！感谢我的家人和朋友。这里要特别感谢Erik、Guy、Mallory、Job、Jenna和Nina在我编写本书期间给予的支持和鼓励，还有Natalie和Marco带给我的美好回忆。最后，感谢CloudSuite的同事，尤其是Corné、Rob、Eméli和Berthold。

谨以此书献给异常风趣、善良而机灵的Melissa和Esmee Hulstein。

目　　录

第 1 章

Java虚拟机



Java虚拟机（Java Virtual Machine，JVM）是一个可用于开发和部署软件的现代平台。顾名思义，最初开发它旨在支持使用Java语言编写的应用程序，但设计Java的人不久就认识到，JVM不仅可运行Java语言，还可利用Java的功能和庞大的类库。

1995年，Sun 公司[①]发布了Java和第一个JVM实现。鉴于其重点是网络应用程序，Java很快就大行其道；它还被设计成可随处运行。开发Java的初衷是用于机顶盒编程，但Sun 公司发现彼时机顶盒市场还不成熟，因此决定同时将这个平台推向台式机。为此，Sun 公司设计了一种独特的二进制可执行格式，并称之为Java字节码。要运行被编译成Java字节码的程序，系统必须安装JVM实现。

本书将简要地介绍5种最流行的JVM语言。通过学习这些语言的基础知识，并动手编写代码，你将能够做出判断，确定哪种语言对你、你的团队和项目来说是最合适的。

我们先来说点实在的，到第2章再深入介绍Java开发包（JDK）和Java类库。当前，可使用的编程语言和平台众多，它们相互争夺市场，因此有必要先来详细地说说JVM向开发人员提供了什么。有鉴于此，本章将介绍如下主题：

- 为何要在JVM上进行开发；
- JVM的常见用途；
- JVM概念简介；
- Java版本；
- 其他JVM语言。

1.1 JVM 实现

需要指出的是，本书只考虑与Oracle Java SE 8（和更高版本）兼容的JVM实现。这个版本可

① Sun公司于2009年4月被Oraele公司收购。——编者注

在台式机、服务器和众多单板计算机（包括尺寸如信用卡的所有Raspberry Pi）上安装。本书使用的是Oracle的JVM实现，你也可使用开源的OpenJDK和IBM的J9 Java SE实现。

本书不涵盖Google发布的用于Android手机和平板电脑的Java平台，因为用于Android的Java版本基于较旧的Java版本。虽然用于Android的Java平台版本越来越新，但它并未提供Oracle Java SE 8的所有功能，且需要使用不同的编译器和工具。另外，Google删除了大量的Java SE API，取而代之的是Google自己开发的不兼容的API。然而，本书介绍的有些语言也可用于Android开发。例如，Kotlin就是一种非常流行的Android开发语言，但本书不会对此展开讨论。

1.2　为何要在 JVM 上开发

当前，可供使用的编程语言和平台众多，为何要在JVM上开发和部署项目呢？因为JVM最初是为Java语言开发的，而近年来，其他语言的拥趸无数次地宣称，Java语言已过时乃至死亡。

近年来，流行的编程语言如走马灯似的更换，而Java犹如常青树，始终处于全球使用最多的编程语言排行榜的前列。

JVM平台为何如此强大呢？下面来看看其中一些最重要的原因：

- 它适应市场的变化，从而确保与时俱进；
- 内置的Java类库非常强大；
- 它有无可比拟的生态系统。

1.2.1　JVM 适应市场的变化

Java于20世纪90年代中期面世，那时的计算机装备的都是单核CPU，内存量也没有几个GB，因为内存条贵得不得了。Java是与时俱进的语言之一：多核CPU面世后不久，Java就通过在多个线程中运行代码来支持多核。但它并没有就此止步，而是在每次推出新版本时，都添加了让并发编程更容易的新类。这种做法到现在依然没有停止。

函数式编程范式大行其道后，Java在核心语言中新增了对lambda和流的支持。虽然Java提供这种支持的时间很晚，但相比于其他流行的语言，其实现更佳。这是因为程序员几乎什么都不用做就能实现多线程。

适应市场变化还意味着有时需要做减法。Java面世时，热点是直接在浏览器中运行Java代码。这些微型程序被称为applet，要求浏览器和系统安装专用的浏览器插件。而现在，市场已将JavaScript作为创建交互式网站的标准语言。有鉴于此，Oracle最近摒弃了applet标准。

1.2.2 Java 类库

在每个Java版本（这将在本章后面更详细地介绍）中，都指定了相应的JVM实现必须提供哪些类。Java SE 8的类库包含大量的类，每个遵循Java SE 8平台标准的JVM实现都必须实现这些类，而不管这种实现是由谁开发的。

这个类库中的类提供了诸如读写控制台窗口、执行文件I/O以及与TCP服务器进行通信等功能，还有很多用于启动和管理操作系统线程的类。更重要的是，还包含定义列表和映射（在有些语言中称为字典）等众多数据结构的类。下一章将详细介绍Java类库中的类。

Java类库是语言设计人员将JVM作为目标开发平台的重要原因之一。有了这个类库定义的数据结构后，他们能够更专注于语言设计，而不用太专注于从头开始打造完整的运行时库。要知道，打造能够与Java类库媲美的、经过全面测试的、多平台运行时系统类库可是一项艰巨的任务。

1.2.3 生态系统

显然，内置的类库不可能涵盖程序员的所有用例。对于内置类库缺失的东西，可求助于其他公司、组织和个人开发的库和工具，以节省开发时间。鉴于很多年来Java都非常成功，其生态系统是无可比拟的，因此很难找到一个这样的平台，即其中可供选择的工具、库、工具包和框架比JVM提供的还要好。

鉴于可供使用的插件库众多，开发人员的开发方向几乎不会受到Java的限制。为说明Java生态系统有多丰富，我们来看看创建Web应用程序时，Java开发人员通常具有的选择空间：

- 创建在JVM应用程序服务器中运行的Web应用程序；
- 为快速获得结果，可使用通用的高级Web框架；
- 为获得更大的控制权，可使用微服务框架来创建应用程序。

场景1：使用JVM应用程序服务器

开发人员可以像企业那样安装基于JVM的应用程序服务器（这可以是开源的，也可以是付费的专用服务器），并使用它来运行应用程序和Web应用程序。这种服务器将负责处理配置问题以及管理数据库连接。

有简单的应用程序服务器，它们只包含运行基本Web应用程序所需的内置API。但也有经过Oracle认证的功能齐备的应用程序服务器，它们提供了大量内置的标准化API，其中包括访问数据库的API、生成或使用XML和JSON文档的API、按SOAP或REST标准与其他Web服务通信的API、确保Web安全的API、向遗留计算机系统发送消息或从这些系统接收消息的API等。

下面是两个最重要的企业开发框架：

- Oracle Java企业版（Java EE）平台，这将在本书后面介绍；
- Spring框架生态系统（其中包括Spring Boot）。

很多应用程序都结合使用了这两种技术。

下面是一些流行的应用程序服务器：

- Apache Tomcat（用于运行简单的Web应用程序）；
- Apache TomEE；
- Red Hat WildFly；
- Oracle GlassFish；
- Red Hat JBoss Enterprise Application Platform；
- Oracle WebLogic。

其中前四个是开源的，而后两个是专用的。

场景2：使用高级的通用Web应用程序框架

第二种选择是使用完整的Web应用程序框架。相比于企业框架，这些框架提供的API通常更高级，它们还提供了内置的**模型–视图–控制器**（model-view-controller，MVC）解决方案，可极大地提高开发人员的效率。

使用这些框架时，开发人员的选择通常受到限制，因为它们只支持为数不多的几个库和工具包。然而，它们支持添加插件来提供其他选择。换而言之，使用这些框架是以放弃一些选择空间来换取更短的开发周期。有些框架要求应用程序运行在JVM应用程序服务器中，而有些提供了自己的HTTP服务器。

在这种框架中，Apache Struts一度非常流行，但现在最流行的可能是Play。

场景3：使用微服务框架

另一种选择是使用现代微服务框架来创建应用程序。这些框架提供了内置的HTTP服务器，可用于运行应用程序，但没有提供其他任何现成的工具和库。在这种情况下，更容易根据需要混合使用不同的库和工具包。

为了采用现代微服务架构，通常将应用程序分成多个独立的Web服务，但这些框架并不要求你必须这样做。

最常用的微服务框架包括Vert.x和Spark Java，其中后者并非是用于Apache Spark大数据平台的。

1.3 常见的用途

前面提供了一些证据，证明了JVM是切实可行的现代软件开发平台，下面来看看一些常见的JVM用途：

- ❑ Web应用程序；
- ❑ 大数据分析；
- ❑ 物联网。

1.3.1 Web 应用程序

JVM非常重视性能，很多人都选择使用它来开发Web应用程序。在设计正确的情况下，应用程序在需要跨越众多不同服务器时的伸缩性极佳。

JVM是一个大家了解得非常清楚的平台，这意味着其行为是可预测的。另外，它还提供了很多工具，可用来调试和剖析有问题的应用程序。鉴于JVM是开源的，完全可以对其内部进行监视。对那些必须同时为数以千计的用户提供服务的Web应用程序来说，这是一个非常重要的优点。

JVM在云计算中扮演着重要的角色。Twitter、Amazon、Spotify和Netflix等很多著名公司都在其基于云的服务的核心部分使用了JVM。

1.3.2 大数据

大数据是当前的一个热点。在数据太大，无法使用传统的数据库进行分析时，可搭建多个数据库集群来处理它们。大数据分析包括查找特定的信息、找出规律、计算统计指标等。

这种数据可能是从Web服务器收集的数据（如已登录用户的单击）、位于制造车间的外部传感器的输出、遗留服务器多年来生成的日志文件等。它们的规模各不相同，但通常多达数TB。

下面是大数据领域两种流行的数据分析技术。

- ❑ Apache Hadoop：负责存储数据以及将数据分发到其他服务器。
- ❑ Apache Spark：使用Hadoop对数据进行流化（stream），以便能够对到来的数据进行分析。

Hadoop和Spark都主要是使用Java编写的。它们都提供了接口，以支持大量的编程语言和平台，其中当然包括JVM。

函数式编程范式致力于创建可在多个CPU内核中安全运行的代码，因此进行Spark或Hadoop编程时，Scala和Clojure等纯函数式编程语言是非常合适的选择。

1.3.3　IoT

当前，能够连接到Internet的移动设备非常普及。鉴于Java最初就是为在嵌入式设备中运行而设计的，因此在IoT领域，JVM也处于优势地位。

对于内存有限的系统，Oracle提供了Java ME Embedded平台。这种平台是专门为不需要标准图形（或基于控制台的）用户界面的商用IoT设备设计的。

对于那些内存还算宽裕的设备，可使用Java SE Embedded版。Java SE Embedded与本书讨论的Java SE很像，在运行完整Linux的环境中，可使用它来提供桌面GUI，以支持全面的用户交互。

Java ME Embedded和Java SE Embedded平台都能够访问Raspberry Pi的**通用输入/输出**（general-purpose input/output，GPIO）针脚，这意味着可通过Java代码来访问这些端口连接的传感器和其他外围设备。

1.4　JVM 概念

要有所作为，JVM开发人员必须熟悉下面这些最重要的JVM概念：

- JVM是一种虚拟机；
- JVM实现大都自带即时（just-in-time，JIT）编译器；
- JVM提供了一些内置的基本类型；
- 除基本类型之外的其他一切都是对象；
- 对象是通过引用类型来访问的；
- **垃圾收集器**（garbage collector，GC）进程将过期的对象从内存中删除；
- JVM大量地使用构建工具。

1.4.1　虚拟机

Java虚拟机是一种虚拟机，这一点显而易见，但必须牢记在心。这意味着从理论上说，在一种计算机上开发的应用程序，可在另一种计算机上运行。

一般而言，代码在32位还是64位的**Java运行时环境**（Java Runtime Environment，JRE）中运行无关紧要。在64位的运行时环境中运行时，应用程序可使用的内存可能更多，但只要应用程序不执行原生操作系统调用或需要数GB的内存，这种差别就无关紧要。



在C等语言中，数据类型的长度取决于原生系统，而Java不存在这样的问题或特色（这是问题还是特色取决于你怎么看）。在JVM中，整型都是无符号的且长32位，而不管运行程序的是哪种计算机平台或系统架构。

最后，需要指出的是，在JVM中运行的每个应用程序都在系统内存中加载自己的JVM实例。这意味着同时运行多个Java应用程序时，每个应用程序都有自己的JVM副本；这还意味着在必要的情况下，不同的应用程序可使用不同的JVM版本。出于安全考虑，不建议在同一个系统中安装不同的JDK或JRE版本；通常最好只安装系统支持的最新版本。

1.4.2 JIT 编译器

虽然没有规定，但所有流行的JVM实现都并非只有简单的解释器：除解释器外，它们还自带了复杂的JIT编译器。

启动Java应用程序时，将首先启动并初始化JVM。JVM启动并初始化后，它将立即开始解释并运行Java字节码。在合适的情况下，解释器将对程序的某些部分进行编译，再将原生可执行代码加载到内存中，并开始执行这些代码而不是经过解释后的Java字节码。这样生成的代码的执行速度通常要快得多。

对代码进行编译还是解释取决于很多因素。对于经常被调用的例程，JIT编译器很可能对其进行编译以生成原生代码。



JIT方法的优点在于，分发的文件可以是跨平台的，且用户无需等待整个应用程序编译完毕。JVM初始化后，应用程序将立即开始执行，而优化是在幕后完成的。

1.4.3 基本数据类型

JVM提供了几个内置的基本数据类型，这是Java未被视为纯粹的OOP语言的主要原因。这些类型的变量不是对象，且始终都包含值。

Java名称	描述和长度	取值范围（含）
`byte`	有符号字节（8位）	–128~127
`short`	有符号短整型（16位）	–32768~32767
`int`	有符号整型（32位）	-2^{31}~$2^{31}-1$
`long`	有符号长整型（64位）	-2^{63} ~$2^{63}-1$
`float`	单精度浮点数（32位）	不精确的浮点值
`double`	双精度浮点数（64位）	不精确的浮点值
`char`	单个Unicode UTF-16字符（16位）	Unicode字符0~655535
`boolean`	布尔值	True/False

请注意，并非所有JVM语言都支持创建基本类型变量，并将其他一切都视为对象。你将看到，这通常不是问题，因为Java类库包含包装基本类型的包装对象，而包含Java在内的大多数语言都

会在必要时自动使用这些包装对象。这个过程被称为**自动装箱**（auto-boxing）。

1.4.4　类

函数和变量都是在类中声明的。即便是应用程序的入口函数（在程序启动时调用的函数`main()`）也是在类中声明的。

JVM只支持单继承模型，即类最多继承一个类。这影响不大，因为使用了名为“接口”的结构来缓解这种影响，你将在下一章看到。接口基本上是一个函数原型（只有函数的定义，而没有代码）和常量列表，编译器要求实现了接口的类必须提供这些函数的实现。类可实现任意数量的接口，但必须提供这些接口定义的每个方法的实现。



本书介绍的有些语言对开发人员隐藏了上述事实。例如，不同于Java，有些语言允许在类声明外面定义函数和变量，甚至允许可执行代码位于函数定义外面。还有些语言支持继承多个类。在内部，这些语言巧妙地规避了JVM的限制和设计决策。

JVM类通常以包的方式进行分组。在下一章，你将看到类是如何组织的。

1.4.5　引用类型

与大多数现代编程语言一样，JVM不直接操作指向对象的内存指针，而使用引用类型。引用变量要么指向特定的类实例，要么什么都不指向。

如果一个引用变量指向特定的对象，就可使用它来调用该对象的方法或访问其公有属性。

如果一个引用变量没指向任何东西，就被称为**空引用**（null reference）。使用空引用来调用方法或访问属性时，将在运行阶段引发错误。对于这个常见的问题，本书介绍的有些语言提供了解决方案。

引用和空引用

请看下面的代码：

```
Product p = new Product();
p.setName("Box of biscuits");
```

假设这里的`Product`是当前程序可使用的一个类。我们创建一个`Product`实例，并让变量`p`指向它。接下来，我们对这个对象实例调用方法`setName`。

JVM没有提供直接访问这个`Product`对象所在内存单元的途径，而只提供了指向该对象的引用。当你使用变量`p`时，JVM将确定为访问这个变量指向的对象，需要访问哪个内存单元。

我们在前述代码片段中添加如下代码行：

```
p = null;
p.setName("This line will produce an error at run-time");
```

可显式地将引用设置为null。请注意，对于在方法内声明的变量，并非必须这样做，因为方法结束时，将自动清理这些变量，但这样做也是完全合法的。现在变量p是一个空引用。下一段将介绍对象实例不再被任何引用变量指向后将发生的事情。

上述代码能够通过编译，但程序运行时，最后一行将引发NullPointerException异常。如果没有提供错误处理功能，应用程序将崩溃。很多现代IDE都力图发现这种情形，并向开发人员发出警告。

1.4.6 垃圾收集器

JVM不要求程序员在创建和销毁对象时手工分配和释放内存块。通常，程序员只需在需要时创建对象即可。

有一个名为GC的进程，它每隔一段时间让应用程序停止执行，并在内存中扫描不再在作用域内的对象（不能被任何对象访问的对象），再将这些对象从内存中删除，并收回释放的内存空间。

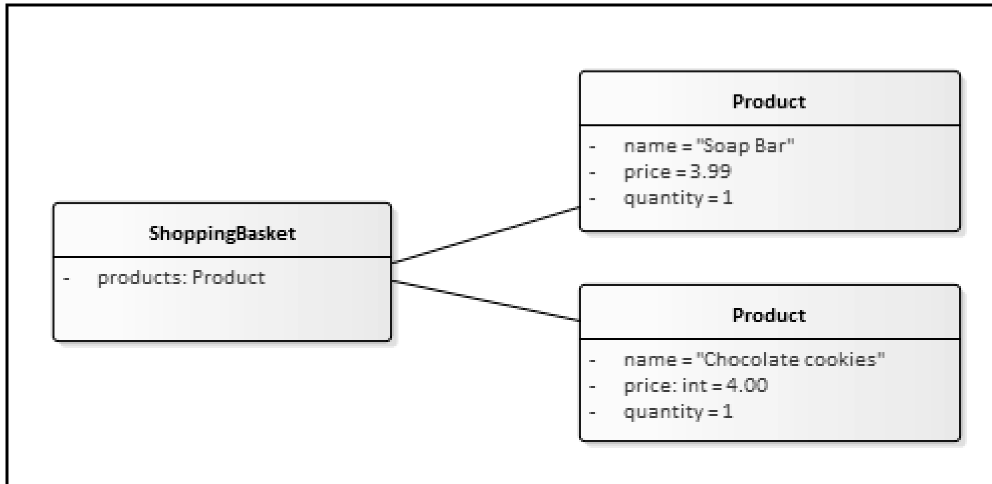
以前，这个进程会导致严重的性能问题，但它使用的算法已得到极大的改进。另外，根据应用程序的需求，系统管理员可配置GC的众多参数，以更好地控制它。

开发人员应始终牢记GC算法。如果你不断地创建大量的对象，并确保它们位于作用域内（即让所有这些对象都是可访问的，如将它们存储在应用程序可访问的列表中），那么迟早会导致内存耗尽，进而引发错误。

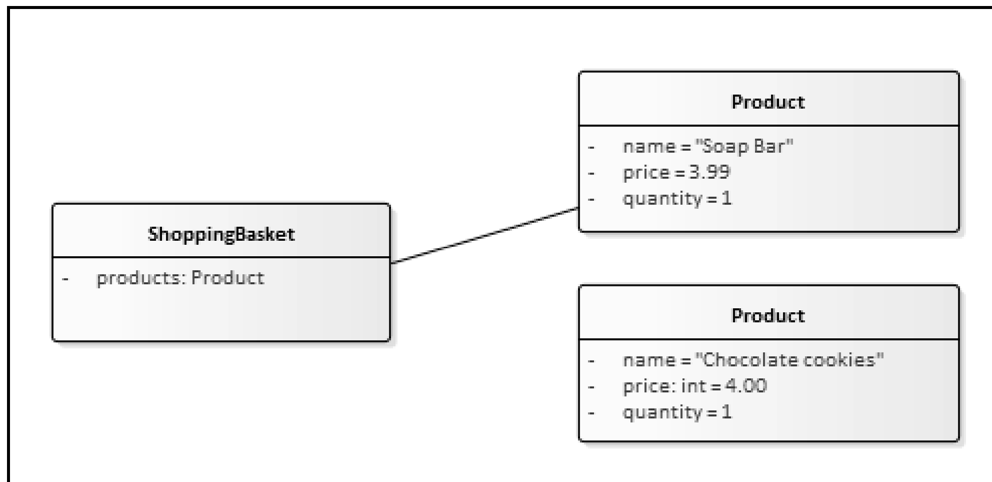
示例

假设你为一个在线商店开发了一个电子商务应用程序，同时假设每位已登录的用户都有一个ShoppingBasket实例，其中存储了该用户已加入到购物车中的商品。

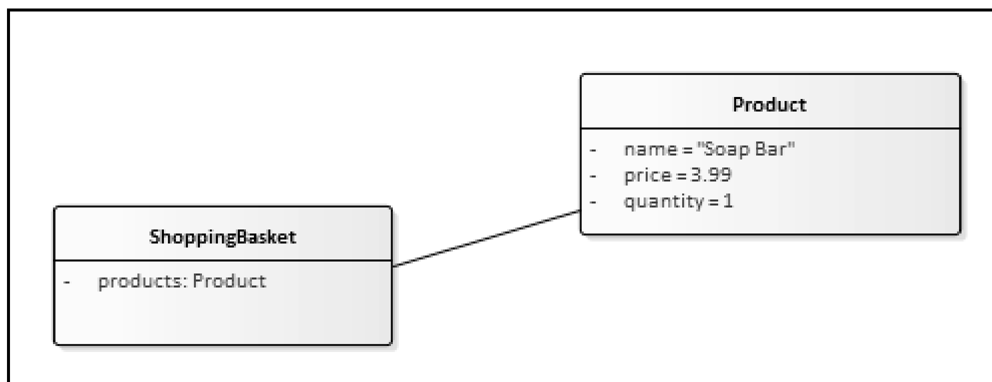
现在假设今天有一位已登录的用户，他打算购买一块香皂和一盒饼干。对于这位用户，应用程序将创建两个Product实例（每件商品一个），并将它们添加到ShoppingBasket的products列表中，如下图所示。



结账前，这位用户发现Amazon也有这样的饼干，但价格低得多，因此决定将其从购物车中删除。从技术上说，应用程序将从products列表中删除相应的Product实例。但这样做后，表示Chocolate cookies的Product实例就成了孤儿对象。鉴于没有任何引用指向它，应用程序再也无法访问它，如下图所示。



过段时间后，JVM的GC启动，它发现应用程序无法访问表示Chocolate cookies的Product对象，因此决定将其删除，从而释放它占用的内存，如下图所示。



为避免GC将对象删除，有多种技巧。其中一个著名的技巧是，在应用程序需要使用大量类似的对象时，将这些对象放在一个对象池（对象列表）中。在应用程序需要对象时，只需从池中取回一个，并根据需要修改它。使用完毕并不再需要该对象时，将其放回到对象池中。由于这些对象始终在作用域内（未用时，这些对象位于应用程序能够访问的对象池中），GC不会销毁它们。

1.4.7　向后兼容

负责维护JVM和Java类库的人深知企业开发人员的需求：现在编写的代码最好以后也能运行。在向后兼容方面，JVM做得很不错；如果你熟悉Python 2和Python 3，就知道情况并非总是如此。

较新的JVM版本能够运行针对较旧的JVM版本编译的应用程序，条件是应用程序没有使用在较新的JVM版本中已删除的API或技术。例如，在Java 8 JVM实例上运行的项目可加载并使用针对Java 6编译的库，但反过来行不通，即在Java 6 JVM实例上运行的应用程序不能加载针对更高版本编译的类。

当然，与其他平台和语言一样，负责维护JDK和Java类库的人必须时不时地摒弃一些类和技术。在向后兼容性方面，JVM虽然存在问题，但总体而言比众多其他的平台和语言要好得多。另外，通常仅当有合适的替代品后，才会将API删除。

1.4.8　构建工具

在项目比较简单的年代，为自动化编译和打包过程，使用的是简单的批文件或操作系统shell脚本文件。随着项目越来越复杂，定义这样的脚本越来越难。另外，对于不同的操作系统，必须编写完全不同的脚本。

不久后，第一套专用的Java构建工具应运而生。它们使用的是XML构建文件，让你能够编写跨平台的脚本。最初，必须编写冗长而繁琐的脚本；但后来，这些工具采用了约定优先于配置的

范式。遵循这些工具指定的约定时，需要编写的代码少得多；但如果你面对的不是默认情形，可能需要花很大的精力来让工具按你希望的做。为自动化构建过程，较新的工具放弃了XML文件，转而提供了脚本语言。

在这些工具中，很多都提供了如下功能。

- 内置的依赖管理器：能够从著名的网络仓库下载附加库。
- 自动运行单元测试，并在测试失败时停止打包。

JDK本身没有提供构建工具，但几乎每个项目都至少使用了下面一个开源的构建自动化工具。

- Apache Ant（没有内置的依赖管理器，使用的是基于XML的构建脚本）。
- Apache Maven（通过使用XML文件引入了约定优先于配置的原则，并使用插件）。
- Gradle（构建脚本是使用Groovy或Kotlin编写的）。

只要使用的是流行的IDE，JVM程序员就无需过多地考虑构建自动化工具，因为所有IDE都能够生成构建脚本。如果要获得更大的控制权，可手工编写脚本，并让IDE根据你编写的脚本来编译、测试和运行项目。

1.5　Java 版本

Java有多个不同的版本，其中每个版本都针对不同的用例。多年来，有些版本的名称发生了翻天覆地的变化，当前版本的名称如下：

- Java标准版（Java SE）
- Java企业版（Java EE）
- Java微型版（Java ME）

1.5.1　Java SE

这是最重要的版本，大家说到Java时，通常指的是就是这个版本。本书只考虑Java SE 平台。

这个版本用于台式机和服务器。另外，你将看到，还有一个嵌入式版本，用于Raspberry Pi的Linux发行版就自带了这种嵌入版本。Java SE自带了完整的Java类库，还包含经典的Swing GUI工具包，而大多数版本还包含较新的JavaFX GUI工具包。



注意：较新的Java SE Embedded更新删除了JavaFX工具包。在Raspberry Pi上安装这个更新后，JavaFX组件将消失。Oracle以开源的方式提供了用于Raspberry Pi的JavaFX，让高阶用户能够下载并编译它。

Java SE主要用于创建独立的控制台应用程序、桌面GUI应用程序和无界面（headless）应用程序，还可用于创建外部库。

1.5.2 Java EE

Java EE建立在Java SE的基础之上，因此要求安装Java SE。它添加了类型众多的API。Java EE应用程序通常运行在JVM应用程序服务器上。本书不会深入介绍Java EE，但时不时会提及它，因为它是Java平台的重要补充，对企业开发人员来说尤其如此。

Oracle网站没有提供独立的Java EE版本，你必须下载与所需Java EE平台版本兼容的应用程序服务器。有些IDE也自带了Java EE应用程序服务器，这将在下一章讨论。

Java EE标准只描述了必须提供的API，而没有指定实现方式。具体如何实现符合标准的API，由与Java EE兼容的应用程序服务器决定。

示例：两款应用程序服务器实现的Java持久化API

Java EE描述了**Java持久化API**（Java Persistence API，JPA）。这是一个**对象关系映射器**（object relation mapper，ORM）API，位于Java对象和关系型数据库（通常是SQL数据库，如Oracle、Oracle MySQL、PostgreSQL等）之间；使用它只需编写几行代码，就可将JVM对象的内容写入数据库，反之（从数据库读取数据并将其加入对象中）亦然。

Oracle提供的Java EE参考实现是一个开源的应用程序服务器，名为GlassFish。GlassFish包含开源项目EclipseLink，该项目实现了JPA标准。Red Hat出品的WildFly也是一款开源的Java EE应用程序服务器，其中包含Red Hat自己开发的ORM开源项目Hibernate，这个项目也实现了JPA标准，但更流行。

如果只使用JPA标准规定的功能，则使用哪种实现无关紧要，但要使用其他功能，选择使用哪种实现就很重要。



如果你不喜欢某个应用程序服务器厂商的设计决策，通常可转而使用其他实现。对于选择空间，JVM开发人员非常在乎！

1.5.3 Java ME

在iOS和Android面世前，Java ME是重要的功能手机和智能手机游戏和应用程序开发平台。iOS和Android都不支持Java ME应用程序，因此它现在已不再是主角。

Java ME包含Java类库的一部分，同时提供了其他一些与移动设备交互的API。Java ME获得了重生，它现在名为Java ME Embedded，可用于商业IoT设备。

1.6 其他 JVM 语言

为推广Java语言和平台，Sun很早就公布了JVM规范。这个文档旨在供那些要自己动手编写JVM实现的开发人员参考，这些JVM实现可能是为那些没有官方JVM实现的平台编写的。这个文档描述了JVM可执行的低级命令、必须提供的数据结构、内存访问规则、Java字节码文件格式.class等。

这个规范的发布让其他语言的设计者能够尝试Java字节码，因此不久后其他语言也能够编译成这种格式。这虽然不是Java设计者的初衷，但Sun（和后来的Oracle）乐见其成。它们是如此地乐见其成，以至于仅仅为方便JVM支持动态语言而添加了新功能。

本节将介绍与其他JVM语言相关的如下主题。

- 为何要放弃Java转而使用其他语言进行JVM开发？
- 在同一个项目中使用多种语言的可能性以及这样做可能带来的问题。
- 使用不同于主项目使用的语言编写单元测试。

1.6.1 为何选择其他语言

考虑到Java语言最初就是为在JVM上运行而设计的，怎么会有人选择使用其他语言来进行JVM开发呢？

开发人员这样做的原因有多个：

- Java是一种非常繁琐的语言；
- 并非所有人都喜欢静态类型语言，且静态语言并非在任何情况下都是最佳选择；
- Java类库缺少一些经常需要用到的类。

1. Java是一种非常繁琐的语言

Java以繁琐著称。经过多年的修订后，Java已不再那么繁琐，但使用很多其他的语言时，完成同样的任务所需的代码更少。

我们来看一个简单的示例。

在Java中，标准的可修改对象通常类似于下面这样：

```
class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  public String getName() {
```

```
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

而使用Kotlin时，只需下面一行代码就能实现同样的功能（以及其他功能）：

```
data class Person(val name: String)
```

这可不是开玩笑。编译这行代码时，Kotlin将自动实现Java示例中定义的方法；实际上，它还会添加其他较为常用的方法。第4章将讨论这些额外的方法。



虽然使用其他语言可极大地提高效率，但Java也没有看起来那么糟糕。所有现代IDE编程工具都能自动生成Java样板代码，如前述示例中的样板代码。为此，你只需按下一个简单的组合键即可。

2. Java并非对所有的人或在任何情况下都是理想选择

虽然Java 8新增了一些重要的函数式编程功能，但从本质上说，它依然是一种静态的命令式语言。并非所有的开发人员都喜欢这种编程风格。如果必须使用静态语言来编写代码，Python或Ruby程序员可能会哭晕在厕所。有鉴于此，有些团队放弃Java转而选择使用其他语言来进行JVM开发。

另外，对于有些问题，使用动态编程语言解决起来要优雅得多；同时对于必须执行复杂并发操作的项目来说，函数式编程风格通常更合适。最后，对于有些哭和框架，通过某些语言来使用让人感觉更为自然。

3. Java类库缺失的类

Java类库是一个庞大的库，但它毕竟是20多年前推出的，因此在有些情况下缺少必要的类。在大多数情况下，功能缺失的问题可通过添加JVM生态系统中免费的开源插件库来解决，但如果选择使用内置了这些功能的语言，不仅更方便，还可节省时间。

例如，对于极其常用的JSON标准，Java SE 8的Java类库就没有提供原生支持。流行的插件库Jackson和Google GSON提供了JSON支持；另外，较新的Java EE平台版本也提供了支持JSON的API。本书介绍的一些语言提供了原生的JSON支持。

另一个问题是，要使用Java类库中的一些常用类，必须编写大量的样板代码。对于Java类库中的众多常用类，诸如Groovy等语言都提供了包装器，让这些API使用起来轻松得多。

1.6.2 在同一个项目中使用多种 JVM 语言

很多语言都能够与Java互操作，因此也能够与其他JVM语言互操作。这是通过尽可能为数据

结构使用标准Java类库并像Java那样编译方法来实现的。

在Java项目中，经常使用其他语言来编写某些类，但这样做可能带来一些问题：

- 构建过程将复杂得多；
- 很多语言要求使用自己的运行时类，这可能带来问题。

1. 构建过程更复杂

使用多种语言时，必须调整构建脚本，这可能导致情况非常复杂。例如，如果Java项目使用了用Groovy编译的类，编译顺序将非常重要，即必须先编译Groovy类，再编译Java代码。如果Groovy代码使用了Java项目中的自定义类，情况将更加复杂。



你将在第11章看到，Groovy比较特殊。Groovy编译器能够编译大部分Java代码，因为从很大程度上说，Groovy语言与Java语言兼容。在没法这样做或这样做不可取时，有一个用于构建工具Apache Maven的编译器插件，使用它可解决构建过程中面临的问题。

一种解决之道是将代码分成多个子项目，并在构建工具中将生成的库作为主项目的需求列出。

有些语言提供了另一种解决方案：它们提供了自定义类，让你能够在Java（或其他JVM语言）中调用这些语言的源代码。被这些类加载时，源代码将动态地编译为Java字节码。其他语言实现了在Java代码中嵌入脚本语言的官方标准；附录A讨论Oracle的JavaScript解释器Nashorn时，将简要地讨论这一点。

2. 语言运行时库

从某种程度上说，这与构建并发症相关。很多JVM语言都要求在编译后的程序中包含支持库，这些库通常定义了语言特有的数据结构以及编译得到的Java字节码调用的内部支持方法。

这通常不是问题，但如果项目的依赖项（或依赖项的依赖项）是使用不同的语言版本编写的，就可能出现问题。如果同一个项目的多个库要求使用同一个运行时库的不同版本，情况将更为复杂：编译或运行项目时，可能出现令人费解的错误消息。

这被称为**依赖恶梦**（dependency hell），但这并非是在同一个项目中使用多种语言特有的现象，而是每个开发人员都应知道的现象。打算在同一个项目中使用多种语言的开发人员还必须明白，语言运行时库可能导致最终程序的规模急剧增大；有些运行时库还包含其依赖项，这将增大出现依赖恶梦的风险。通常，语言的文档或官网都详细说明了其依赖项。



与很多框架设计者一样，很多语言设计者都对这些问题心中有数，并采取了措施降低这些问题出现的风险。例如，对于自己使用的较流行的依赖项，他们进行了重命名，以防发生类名冲突。

1.6.3　使用另一种语言编写单元测试

使用其他语言编写的单元测试来对Java代码进行测试很常见。正如你在本章前面看到的，相比于Java，使用其他语言编写的代码可能紧凑得多，这些语言非常适合用来编写小型、具体而易于理解的单元测试。

鉴于仅在运行单元测试时才会用到这些语言的运行时库，因此无需在编译得到的主项目中包含它们。



你将在第11章看到，在这样的情形下，非常适合使用Groovy。Groovy提供了一些便利的单元测试编写功能，其中包括内置的`assert`语句，这种语句在传入的值与期望值不同时将打印非常详尽而易于理解的输出。。

1.7　小结

本章非常简略地描述了JVM。首先介绍了JVM向开发人员提供的功能、JVM的常见用途以及最重要的JVM概念；接下来探索了Java版本；最后介绍了其他JVM语言，以及开发人员放弃Java转而使用其他语言进行JVM开发的原因。

在下一章，我们将安装并详细介绍JDK，还将详细介绍Java类库并安装其他开发工具，为动手开发做好准备。

第 2 章

Java虚拟机开发

2

本章深入介绍Java虚拟机（JVM），重点是每个JVM开发人员都必须知道的概念，而不管他选择使用的是哪种编程语言。本章涵盖如下主题：

- Java开发包（JDK）；
- 使用包来组织类；
- Java类库；
- 从命令行运行JVM应用程序；
- 安装Eclipse集成开发环境（Eclipse IDE）。

本书涵盖Windows、macOS和Linux（Ubuntu）操作系统，但展示路径时通常使用Windows风格。如果你使用的是macOS或Linux，务必按相应操作系统的规则修改路径。

2.1 JDK

要进行JVM开发，必须安装JDK。JDK包含Java运行时环境（Java Runtime Environment，JRE）、Java编译器以及各种开发工具（本章将介绍其中的一些）。即便你的大部分开发工作都将使用其他语言而不是Java来完成，也强烈建议你安装完整的JDK，因为很多重要的开发工具都必须安装完整的JDK才能运行。另外，你迟早都将用到一些只有JDK才有的工具。

安装较新的用于Raspberry Pi的Linux发行版时，如果使用默认的Raspian安装选项，将自动安装Java SE Embedded 8 JDK，但提供的版本通常不是最新的。其他主要的操作系统默认安装的JDK如何不得而知。本节介绍如下与JDK相关的主题：

- 安装JDK（Windows、macOS和Linux）；
- 探索JDK；
- JRE。

2.1.1 安装 JDK

这里只介绍如何安装Oracle的JDK 8实现。如果你已安装与Java SE 8平台完全兼容的JDK实现，包括开源的OpenJDK 8或IBM的J9 JDK 8，通常也完全可行，因此你可以跳过本节。



非Oracle的JDK实现并不一定包含本书讨论的所有功能，必要时我们将指出例外情况。

无论你使用的是哪种操作系统，都必须创建一个名为`JAVA_HOME`的环境变量，并让它指向JDK安装目录（在非开发计算机中为JRE所在的目录）。对于本书涵盖的操作系统（Windows、macOS和Linux），我们将说明如何创建这样的环境变量。很多重要的JVM工具都要用到这个变量，包括构建工具和应用程序服务器。本节将介绍如何完成如下工作：

- 下载JDK；
- 在Windows系统中安装JDK；
- 在macOS系统中安装JDK；
- 在Linux系统中安装JDK；
- 下载Javadoc API文档。

1. 下载JDK

Oracle提供的用于Windows、macOS和Linux的JDK实现可从Oracle网站下载。对于有些平台，只有64位的JDK，而对于其他平台，有32位和64位的。

使用你喜欢的浏览器访问Oracle的Java主页（http://www.oracle.com/java），如下图所示。



要下载JDK，需执行如下步骤。

(1) 本书编写期间，该网页包含一个Java for Developers，单击该按钮将进入Software Downloads部分。

(2) 在列表中找到并单击Java SE（包含JavaFX）。如果它旁边有链接Early Access，请注意不要单击该链接。

(3) 进入Java SE Downloads页面。单击JDK下方的Download按钮。

(4) 根据你使用的操作系统平台和体系结构，找到并单击相应的版本。

(5) 如果你同意了许可条款，将下载相应版本的JDK。

2. 在Windows系统中安装JDK

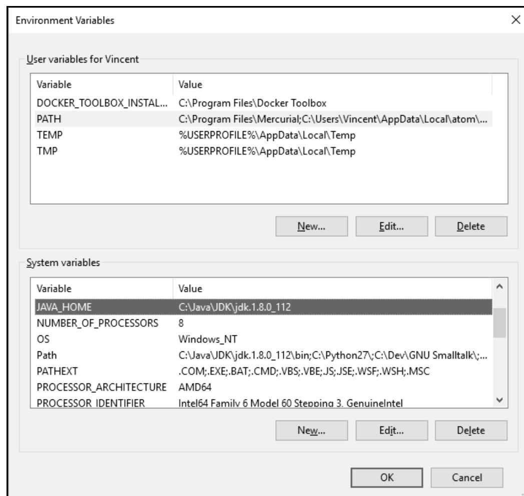
对于Windows操作系统，JDK有32位和64位的版本。你只需运行下载的可执行文件并按说明做即可。请将JDK安装路径记录下来，因为后面要用到。

请使用默认设置，这将随JDK安装JRE。强烈建议确保JRE和JDK的版本同步，因此使用JDK安装程序安装JRE是最佳选择。Oracle指出JDK始终是在系统级安装的，因此其他所有用户都可使用它。

安装完毕后，需要添加或修改一些环境变量。这里介绍如何在Windows 10中这样做，如果你使用的是其他较新的Windows版本，做法应与之类似。

(1) 右击Windows“开始”按钮并选择“系统”，在出现的窗口中，单击左边的“高级系统设置”。

(2) 这将打开“系统属性”对话框。单击“环境变量”按钮，这将打开如下图所示的“环境变量”对话框：



(3) 在这个对话框底部的“系统变量”部分查找变量`JAVA_HOME`。如果没有找到，就单击按钮“新建”，否则对既有的`JAVA_HOME`进行编辑。

(4) 将变量名设置为`JAVA_HOME`，并将值设置为JDK安装目录的完整路径，再单击“确定”按钮关闭打开的对话框。

(5) 找到既有的变量`Path`，并在其中添加JDK安装目录下的子目录bin的完整路径；别忘了用字符`;`将不同的目录分隔。

为验证安装，可执行如下步骤。

(1) 打开一个命令提示符窗口（为此，可单击“开始”按钮，输入`cmd`并按回车键）。

(2) 输入`javac -version`并按回车键。

你将看到一个版本号，它与下载的JDK版本相同。如果不是这样，请核实你是否正确地修改了环境变量，并确保你新打开了一个命令提示符窗口，而不是使用以前打开的命令提示符窗口。

3. 在macOS系统中安装JDK

请注意，要安装JDK，使用的macOS版本必须是较新的。在本书编写期间，JDK 8要求macOS版本至少为10.8（Mountain Lion）。

在macOS系统中安装JDK很容易，只需双击下载的映像（.dmg）文件，再在出现的Finder窗口中双击包图标，然后按说明做即可。与Windows系统中一样，在macOS系统中，JDK也是在系统级安装的，因此可供所有用户使用。

安装完毕后，需要确保新安装的JDK是默认使用的JDK。macOS支持同时安装多个JDK版本，你可在不同的版本之间切换，但只有一个版本处于活动状态。要切换到新安装的JDK版本，最简单的办法是打开位于你的Home文件夹中的文件.bash_profile（请注意这个文件名以句点打头），并在其中添加如下行：

```
export JAVA_HOME="$(/usr/libexec/java_home -v 1.8)"
```

要验证安装，可执行如下操作：

- 打开一个新的Terminal窗口；
- 在其中输入`javac -version`并按回车键。

显示的版本号应与你下载的JDK版本相同。

4. 在Linux系统中安装JDK

Linux JDK有32位和64位版本，可以如下格式下载它们：

- 下载压缩的.tar.gz文件，用于手动安装；

- 下载RPM包管理器（RPM Package Manager）文件（.rpm），用于在支持这种打包格式的Linux中安装JDK。

Oracle认证了多个可安装JDK的Linux发行版。在本书编写期间，各种Oracle Linux、Red Hat和Ubuntu都通过了认证。即便你使用的不是这些Linux发行版，也并不意味着你的计算机不能运行JDK和JVM，而只是意味着没有得到Oracle的官方支持。

本节只介绍如何在Ubuntu系统中安装JDK。Ubuntu没有提供原生的RPM格式支持，因此如果你使用的是Ubuntu，推荐下载.tar.gz文件。在Linux系统中，虽然并非必须在系统级安装JDK，但这里将这样做。如果你使用的Linux发行版支持RPM或不支持后面用到的一些命令，请参阅Java SE Downloads页面中的链接Installation Instructions。

打开一个新的Terminal窗口，切换到下载的.tar.gz文件所在的目录，并输入如下命令：

```
su
tar xvfz jdk-VERSION-linux-x64.tar.gz
ls
mv jdk1.VERSION /usr/local/
```

对这些命令说明如下。

- 这要求你有系统根密码。如果没有这样的密码，请将`su`替换为`sudo -s`，但仅当你有根权限时，命令`sudo -s`才管用。
- 必须将`VERSION`替换为你下载的JDK的版本号。
- 这里假定平台是64位的（x64），如果你下载的是32位版本，请将`x64`替换为`i586`。

请注意，在最后一个命令中，你移动的是从下载的文件.tar.gz解压缩得到的目录（而不是这个文件本身）。另外，VERSION格式的目录不同于下载的文件。请将/usr/local/jdk1.VERSION的完整路径复制到剪贴板或记录下来，因为下一步需要用到。

现在来设置环境变量`JAVA_HOME`。我们希望Terminal为每位用户自动加载这个环境变量：

```
nano /etc/profile
```

滚动到这个文件末尾，并添加如下内容（将`JAVA_HOME=`后面的内容替换为前面复制到剪贴板中的路径）：

```
JAVA_HOME=/usr/local/jdk1.VERSION
export JAVA_HOME
```

按Cltrl + X并选择Y来保存所做的修改，再按回车确认文件名。

最后，你需要向Ubuntu注册JDK和JRE命令。通过在Terminal窗口中输入如下命令，将为两个最重要的命令创建合适的符号链接：`java`（用于运行JVM应用程序）和`javac`（用于运行Java编译器）：

```
. /etc/profile
update-alternatives --install "/usr/bin/java" "java" $JAVA_HOME/bin/java 1
update-alternatives --install "/usr/bin/javac" "java" $JAVA_HOME/bin/javac
1
```

其中第一个命令重新加载修改后的文件/etc/profile，以便能够使用变量`JAVA_HOME`。请注意这个命令开头的句点。

2



如果要使用JDK子目录bin中的其他命令，对于每个这样的命令，都必须以根用户的身份在Terminal中执行相应的`update-alternatives`命令。

执行命令`exit`两次将Terminal窗口关闭，再重新打开Terminal窗口（这次无需根权限），并输入如下命令来验证安装：

```
javac -version
```

如果一切顺利，将出现与下载的JDK版本相同的版本号。

5. 下载API文档

Oracle提供了完整的Java类库API在线文档。对于Java 8，要查看这种文档，可访问https://docs.oracle.com/javase/8/docs/api/。

在本地有该文档的副本很有帮助，Oracle认识到了这一点并提供了下载。

- 访问Java SE Downloads页面（有关如何访问这个页面，请参阅“下载Download”一节）。
- 找到Additional Resources部分，并单击Java SE 8 Documentation旁边的Download 按钮。
- 如果你接受许可协议，就可下载相应的ZIP文件。

将这个文件解压缩到方便的位置，再使用你喜欢的浏览器打开子目录docs中的文件index.html。



如果你要使用工具包JavaFX创建桌面GUI应用程序，也应从这个下载页面下载JavaFX API文档。

2.1.2 探索 JDK

JDK最重要的组件如下：

- `java`（用于运行编译后的JVM应用程序，即便这种应用程序不是使用Java编写的）；
- `javac`（Java语言编译器）。

JDK并非只包含这两个组件；本节将介绍JDK的目录结构并概述目录bin中最重要的命令。

1. 目录结构

要熟悉JDK，了解其目录结构大有裨益。下面以图形方式概述了JDK安装目录包含的子目录：



下表更详细地说明了这些子目录。

目　录　名	描　　述
bin	子目录bin包含JDK提供的所有可执行命令，接下来将讨论其中最重要的命令
db	与JavaDB组件相关的东西都存储在这里。JavaDB是Oracle支持的数据库项目Apache Derby，这是一个开源的基于文件的关系数据库系统，提供了强大的SQL支持。它完全是使用Java实现的。JDK 9删除了这个组件，但感兴趣的读者依然可以从Derby网站（https://db.apache.org/derby/）下载它
include	这个目录是为高级程序员准备的，其中包含供C语言编译器使用的头文件，可在Java代码中使用它们来调用随平台或操作系统而异的原生代码，或反之
jre	这里存储了所有与JRE相关的文件，包括Java类库。请注意，目录jre/bin中的所有命令也都包含在JDK安装目录下的子目录bin中
lib	这里存储了供一些开发工具使用的库

2. JDK命令

目录bin包含JDK提供的主要的命令行驱动命令。下表列出了其中最重要的命令，但并非所有这些命令都会在本书中做进一步的讨论。

可执行的命令	描　　述
`java`	加载一个JVM实例并启动在命令行中指定的程序，本章后面将更详细地讨论它。在Windows系统中，它将在运行应用程序时打开一个控制台文本窗口
`javac`	这是Java语言编译器
`javadoc`	从Java源代码文件中提取并生成文档，将在下一章简要地讨论
`javap`	对编译后的Java代码进行反汇编，生成类似于Java字节码的易于理解的文本格式
`javaw`	只有Windows版JDK和JRE提供了这个命令，它与命令`java`相同，但不会打开额外的窗口。如果启动的应用程序有桌面GUI，应用程序仍将在独立的窗口中打开
`jar`	用于创建JAR归档文件、从JAR归档文件中提取数据以及在其中添加文件的工具。JAR归档文件将在本章后面更详细地介绍
`jarsigner`	通过添加数字签名来保护JAR文件。如果JAR文件的数据被修改，但签名没有相应地更新，文件将被视为无效的
`jdeps`	输出编译得到的.class文件或JAR文件的依赖信息
`jjs`	激活Oracle Nashorn的交互式解释器shell。Oracle的JavaScript解释器为Nashorn，将在附录中讨论

请注意，目录bin还包含这里没有列出的其他命令，它们大都仅供高级用户使用。

3. GUI监视工具

子目录bin中包含上表未列出的三个工具，这里有必要说一说。不同于其他命令，这些命令提供了完整的桌面GUI：

- ❑ Java VisualVM；
- ❑ Oracle Mission Control；
- ❑ JConsole。



只有Oracle的JDK实现包含这三个工具。开源实现OpenJDK不包含Oracle Mission Control，而IBM J9 JDK不包含上述任何工具。

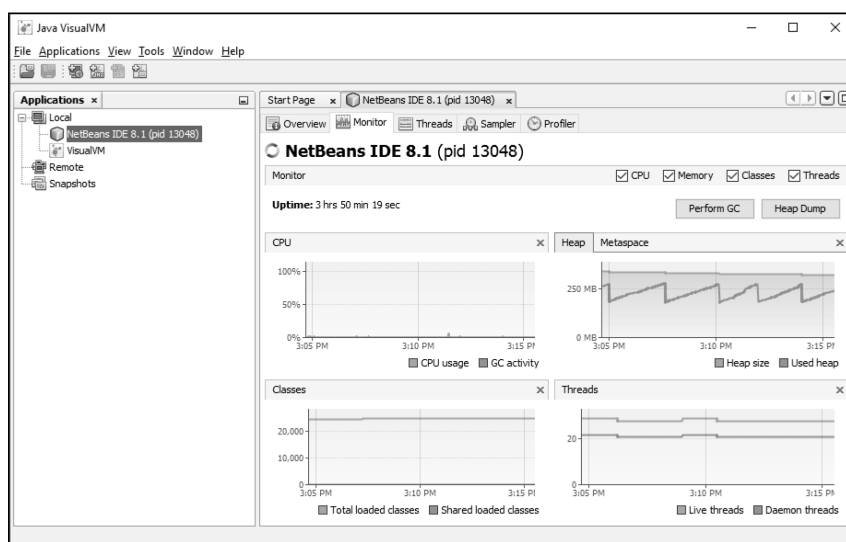
(1) Java VisualVM

在8.0版之前，Oracle JDK和OpenJDK都包含Java工具VisualVM。VisualVM是一个开源工具，用于监视所有运行JVM应用程序的JVM实例，你可以通过安装插件来进一步改进其内置功能。如果你使用的是Oracle JDK 9或OpenJDK 9，可以从https://visualvm.github.io/index.html下载这个开源工具。

要使用默认设置启动VisualVM，可执行如下命令：

```
jvisualvm
```

这将首先出现一个启动窗口，过段时间后再出现VisualVM主窗口。VisualVM不仅能够连接到网络服务器上运行的JVM实例，还能连接到本地运行的JVM实例。在下面的屏幕截图中，监视的是本地运行的NetBeans IDE实例：



要设置JVM实例以便进行远程监视，需要费点劲，这不在本书的讨论范围内，但JDK文档的VisualVM部分做了详细说明。



实时监控将消耗大量的系统资源。仅当在开发环境中才应考虑对本地进程进行监控。远程监控消耗的服务器资源要少些，但也相当可观。

(2) Oracle Mission Control

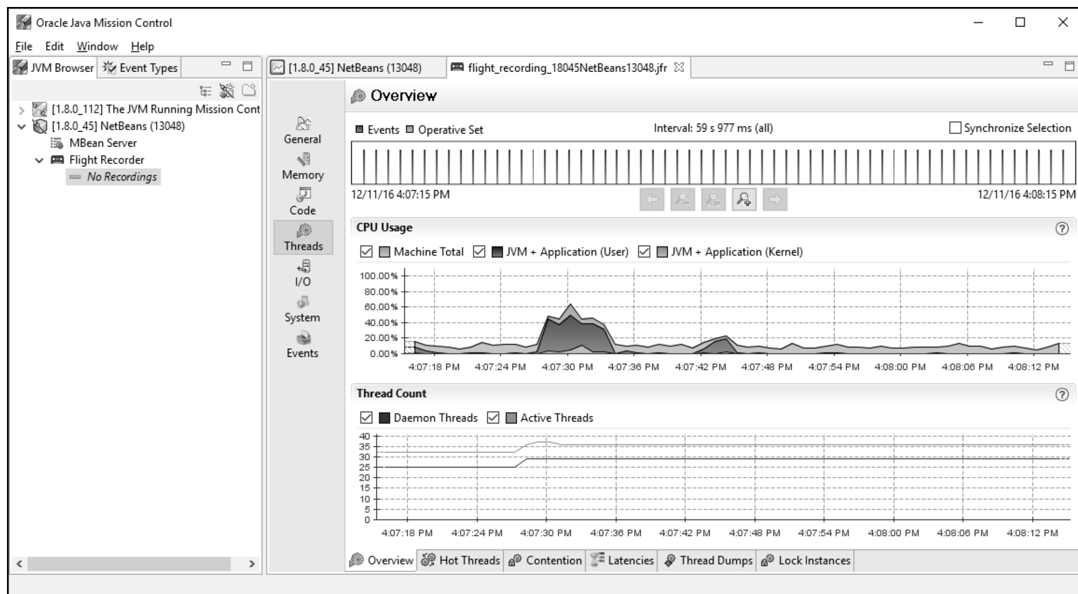
较新的Oracle JDK版本都自带了Oracle Mission Control，这也是一个监视JVM实例和应用程序的工具。Oracle Mission Control提供的用户界面比Java VisualVM还好，但很多功能都与VisualVM类似，包括实时监控正在运行的JVM实例和应用程序。

Oracle Mission Control是款专用软件，其许可条款比较复杂。其大部分功能都可在开发和生产环境中免费使用，但其独特的功能Java Flight Recorder只能在开发环境中免费使用。要在生产环境中使用Java Flight Recorder，必须有付费从Oracle获得的许可密钥。

要运行Oracle Mission Control，可执行如下命令：

```
jmc
```

通过使用Java Flight Recorder，可记录JVM在指定时段内发生的事件。记录过程结束后，可对记录的所有数据进行分析。Java Flight Recorder的优点在于，开销比Oracle Mission Control和VisualVM的实时监控功能都低得多，因此在生产系统中使用它要安全得多（但别忘了其许可条款）。在下面的屏幕截图中，使用Flight Recorder对NetBeans IDE进程监视了1分钟。



(3) JConsole

JConsole是最古老的监视工具，JDK在很久前就提供了它。相比于JConsole，Java VisualVM和Oracle Mission Control提供的功能都更丰富，GUI也更友好，因此建议你不要使用JConsole，而使用其他工具。

如果你一定要使用JConsole，可使用如下命令来启动它：

```
jconsole
```

2.1.3 JRE

如果只想在计算机上运行Java程序，可只安装JRE，这将安装用于启动JVM实例的命令`java`以及完整的Java类库。JRE用于启动使用Java或其他语言编写的应用程序。





使用默认设置安装JDK时，将同时安装JRE。仅在不需要开发工具的计算机上，才需要单独下载并安装JRE。

使用较旧OS X（在Apple将其重命名为macOS之前）的Mac计算机通常预安装了Java运行时，但现在情况不再如此，因为Oracle已从Apple手工接管了macOS Java SE实现的开发工作。

Oracle网站提供了两个版本的Java SE 8 JRE：

- JRE
- Server JRE

JRE用于32位或64位最终用户台式机（请注意，并非所有平台都有32位版本），而Server JRE用于服务器，通常由高级系统管理员安装。Server JRE只能用于64位系统，不包含安装程序和浏览器插件，但包含前面讨论的JVM监视工具。

2.2　使用包组织类

所有JVM语言都定义了其创建类和实例化对象的语法，但它们生成的类文件最终都将在JVM上运行。为了能够在JVM上运行以及与使用其他JVM语言编写的类互操作，必须遵循JVM在类组织方面的要求。本节将讨论如下主题：

- 包；
- 选择包名；
- 包名举例；
- 全限定类名。



要明白Java类库的组织方式以及如何从命令行运行JVM应用程序，必须对包有所了解。这两个主题都将在本章讨论。

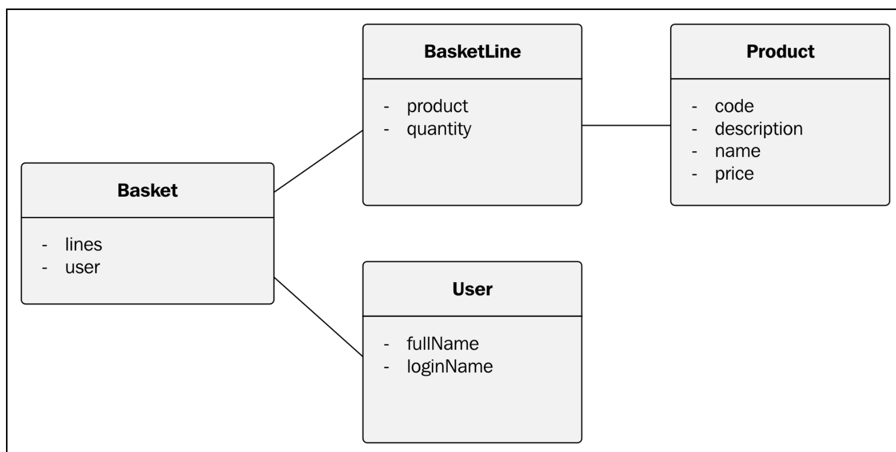
2.2.1　包是什么

本书介绍的语言大都支持将类组织成包。位于同一个包中的类构成了一个独特的命名空间，这是JVM最重要的特征。有些语言本身不支持将类组织成包，但支持引用包中的类。



一个这样的例子是Oracle的JavaScript解释器Nashorn。JavaScript语言本身不支持包，但Nashorn能够导入放在包中的Java（或其他兼容）类。

为演示如何将类组织成包，下面再来看看电子商务应用程序中购物车的实现，但这里的示例将更成熟些。



JVM提供了一种对类进行组织的方式——将它们放在包中。通过这样做，可将主题相同的类

编组。就刚才的示例而言，将涉及的类放在下面这些包中比较合理。

包	包中的类
basket	Basket、BasketLine
product	Product
user	User

之所以选择这种组织方式，是因为我认为这些类的主题分别为购物车、商品和用户。另外，我预计未来Basket、Product和User类将被其他类使用，而BasketLine类只会在Basket类内部使用。

通过使用包，不仅可让大型项目的结构更容易理解（这一点你将在稍后看到），还可通过使用访问修饰符对其他包中的类隐藏类成员。

要高效地使用包，必须了解几乎所有JVM项目都遵循的约定。虽然上表选择的包名完全合法，但给包命名时应遵循一定的约定。

2.2.2　选择包名

包名必须遵守如下几个规则。

- 包名可包含句点；实际上，句点用于分隔名称中的不同元素。
- 包名中的元素可包含字母、数字和下划线。
- 包名中的每个元素都必须以字母打头，而不能以数字打头（数字只能跟在字母后面）。
- 包名中的元素不能是Java保留的关键字，这包括基本类型的名称（如int、short）以及内置关键字（如class、for和final）。

有一些大多数重要项目都遵守的通用命名约定。在所有项目中，都强烈建议你在给包命名时遵守如下约定。

- 整个包名都为小写。
- 包名以如下URL打头。
  - 公司网站的URL。
  - 项目网站的URL。
  - 项目公开源代码仓库的URL。
  - 个人主页或博客的URL。
- 你可制定自己的约定，以便将不同的类区分开来，避免不同的项目发生命名冲突。为此，可在包名中添加部门名称、办事处名称或项目名。
- 如果包名包含非法元素，可在前面或后面添加下划线使其合法，或将非法字符替换为下划线。

2.2.3　包名举例

假设你受雇于JVM University，要为它开发一个在线Web编程类，而这个类可通过http://www.example.com/jvm-university来访问。

在这种情况下，合法的包名如下：

- `com.example.jvm_university.class_.web_programming`
- `com.example.jvm_university.webprogramming`
- `com.example.jvm.university.web_programming`

2.2.4　全限定类名

在有些情况下，必须使用全限定包名，即以完整的包名打头的类名。来看一个示例：

```
package com.example.jvm.university.web_programming;
class Application {
}
```

对于这个类，全限定类名为`com.example.jvm.university.web_programming.Application`。

2.3　Java 类库

Java类库也被简称为Java API，这是随Java SE平台分发的大量预置类。下面是Java类库包含的一些重要主题：

- 常用数据结构的定义和实现
- 控制台I/O
- 文件I/O
- 数学运算
- 联网
- 正则表达式
- XML的创建和处理
- 数据库访问
- GUI工具包
- 反射

这里无法全面介绍Java类库，而只提供一些API示例，让你大致知道去哪里寻找所需的类。介绍具体的类之前，我们先来看看Java类库的主要组织结构，这包括如下主题：

- Java类库的组织结构

- 包概述
- java.lang包中的重要类
- 集合API，具体地说是java.util.ArrayList和java.util.HashMap

2.3.1 Java 类库的组织结构

Java类库中的所有类都放在包中。最重要的包的名称都以如下两项内容打头：

- java
- javax

这主要是历史原因导致的。声誉良好的现代Java SE实现都实现了这两个包中的类，但还有其他的公有类。一些杂项类放在以org打头的包中，如org.w3c和org.xml，但本书不会介绍这些类。

厂商可在库中添加自己的类，在Oracle的实现中，这些类位于名称以com.sun、sun或com.oracle打头的包中。然而，建议你使用附加库而不是这些包中的类。

2.3.2 包概述

为了让你对类的分组情况有大致了解，这里简要地介绍一下Java类库中最重要的包。这只是个引子，旨在让你熟悉Java类库的结构，因此并没有列出所有的包。

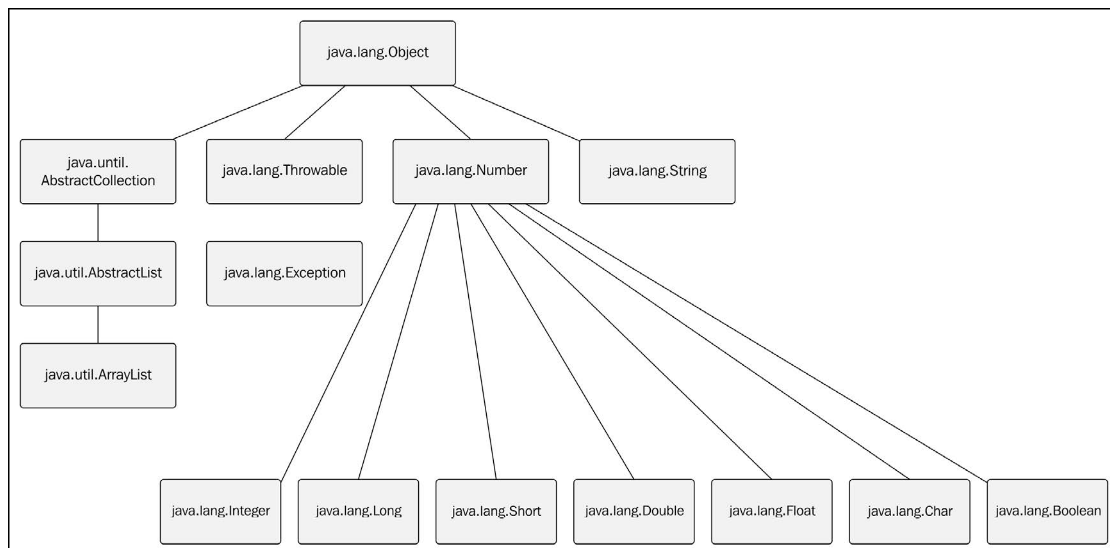
包	描　述
java.lang	这个包中的类是最重要的，其中包括String和StringBuilder类、基本类型包装类、线程化类以及所有类的祖先——Object
java.lang.reflect	提供了反射API。反射让你能够动态地检查类，以获悉方法和变量的名称、调用方法以及读写属性
java.uil	最重要的包之一，包含实现集合、日期和时间、国际化等的类
java.util.concurrent	包含用于并发编程的类
java.io java.net java.nio	包含与操作系统、文件和网络I/O相关的类，还包含字符集编码/解码类
java.math	提供了BigDecimal类。相比于基本类型float和double以及BigInteger类，这个类要精确得多，它能存储的整数值比基本类型int和long大得多
java.xml	包含XML处理类
java.sql javax.sql	包含用于访问JDBC数据库系统的类
java.awt	抽象窗口工具包（Abstract Window Toolkit）——最早的Java GUI工具包，位于操作系统原生GUI和JVM之间
javax.swing	包含Swing GUI工具包类，这些类是建立在AWT工具包的基础之上的。一个不同于AWT的重要之处是，其所有GUI控件都是使用Java代码实现的
javafx	JavaFX GUI工具包类，这是一款非常时髦的3D加速图形产品

2.3.3 java.lang 包中的重要类

对JVM平台来说，`java.lang`包中的类至关重要，因此这个包中的很多类在本书后面经常会被提及。本节旨在提供一些背景信息，而不是要取代Java API文档。本节将介绍下面这些类：

- Object类（`java.lang.Object`）；
- String类（`java.lang.String`）；
- 基本数据类型包装类（`java.lang`包中的Integer、Long、Short、Char、Float和Double）；
- 异常和错误（`java.lang.Exception`和`java.lang.Error`）。

下面是这里要讨论的类组成的类层次结构图：



本书后面还将讨论Java类库中众多其他的类。

1. Object类（java.lang.Object）

`java.lang`包中的Object类是其他所有类的基类；在JVM中，只有它没有父类。在Java语言中，没有显式地继承其他类的类都隐式地继承`java.lang.Object`类。

- **Object类的重要方法**

下表列出了`java.lang.Object`类中最常用的方法。

方法名	返回类型	描 述
toString()	String	返回对象的文本描述
equals(Object object)	boolean	指出传入的对象是否与当前对象相等，它使用多种规则来判断相等性，这将在后面讨论
hashCode()	int	计算当前对象的散列值，将在后面介绍集合API时讨论

Object（以及每个JVM对象）提供的重要方法之一是toString()，它返回当前对象实例的文本描述。Oracle提供的默认实现是返回全限定类名以及十六进制格式的hashCode()结果，但建议在自定义类中重写这个方法，以提供更易于理解的描述。

集合API大量地使用了方法equals()和hashCode()，后面将更详细地介绍它们的用法。

有关java.lang.Object类的完整方法列表，请参阅API文档。

2. String类（java.lang.String）

java.lang.String类表示JVM中的字符串。这是一种不可修改的对象，这意味着修改String对象时不会影响原始对象，而是生成一个包含修改后内容的新字符串。字符串在内部都是使用UTF-16编码存储的。

这个类将在下一章更详细地介绍。本书介绍的有些语言有自己的字符串类，这些类包含额外的便利方法和独特的功能。通常，JVM语言会在幕后透明地在这两种字符串类型之间进行转换。

3. 基本类型包装器类（java.lang包中的Integer、Long、Short、Char、Float、Double类）

并非所有的Java API都能够使用JVM的内置基本数据类型。如果需要的是基本类型包装类，而你传递的是基本数据类型变量，编译器将自动创建指定包装类的实例。反之亦然，即在需要的是基本数据类型变量，而你传递的是包装类对象时，编译器将把包装类对象的值赋给基本数据类型变量。这个过程被称为**自动装箱**（autoboxing）。



并非所有的JVM语言都支持自动装箱，但大多数流行的JVM语言都支持，包括本书介绍的所有语言。

与String类一样，这些类也都是不可修改的。调用任何修改值的方法时，都将创建并返回一个包含新值的实例。

前一章说过，有些JVM语言遵循“一切皆对象”的面向对象编程规则，不支持基本数据类型，因此使用包装类来处理基本类型值。

- 自动装箱示例

下面的代码将一个基本类型int值赋给一个java.lang.Integer引用变量：

```
int primitiveInt = 42;
Integer wrappedInteger = primitiveInt;
```

这将创建一个包装了`int`值42的`Integer`对象，与显式地创建一个`Integer`实例等效：

```
Integer wrappedInteger = new Integer(42);
```

下面的代码将两个`Integer`实例传递给一个将两个`int`值作为参数的API，结果符合预期：

```
System.out.println("Hello world".substring(new Integer(0),
                     new Integer(5)));
```

这将打印Hello。

4. 异常和错误（**java.lang.Exception**和**java.lang.Error**）

JVM开发人员必须知道JVM是如何管理运行阶段错误的。鉴于所有的语言都有自己的运行阶段错误处理机制，这里不介绍错误是如何处理的，而只说说发生运行阶段错误时的情况。

在方法中发生运行阶段错误时，将创建并引发一个异常或`Error`对象。在Java语言中，这是使用关键字`throw`实现的。下面是一个引发通用异常`Exception`的示例：

```
throw new Exception("Oops!");
```

Java有很多继承`Exception`或`Error`类的内置类。创建自定义异常类时，应考虑可重用哪个既有的异常类。例如，如果方法要求传入的引用不能为空，应在传入的参数为空时引发异常`java.lang.NullPointerException`；所有Java API在用户将空引用传递给不支持它的方法时都这样做。



注意：如果你仔细研究本节开头的类图，将发现`Exception`和`Error`都继承了`Throwable`类。`Throwable`是可引发的对象，但通常使用`Exception`和`Error`的子类，因为它们使用起来更方便。

`Exception`和`Error`类的不同之处如下。

- 在程序很可能能够妥善地处理错误并继续运行时引发异常。
- 发现根本没有想到的问题时引发错误。很多错误都是由JVM本身引发的。

异常或错误（以下简称异常）被引发时，JVM将查看引发异常的方法。如果它包含能够处理错误的错误处理程序，就把控制权交给错误处理程序。如果这个方法不能处理任何错误（或当前错误），就检查方法调用方是否包含错误处理程序。这个过程将不断重复，直到找到能够处理当前错误且不引发新错误的方法，或者进入第一个方法调用。在第二种情况下，JVM实例将崩溃并生成类似于下面的栈跟踪：

```
Exception in thread "main" java.lang.Exception: Oops
 at ExceptionDemo.method3(ExceptionDemo.java:37)
```

```
at ExceptionDemo.method2(ExceptionDemo.java:33)
at ExceptionDemo.method1(ExceptionDemo.java:29)
at ExceptionDemo.main(ExceptionDemo.java:25)
```

很多语言都在生成的Java字节码中包含源代码文件名和行号，这样可以在栈跟踪中包含源代码行号，让栈跟踪更容易理解。

Java有严格的异常引发规则，因此并非每个类都能够引发所有的异常；在这方面，很多其他的JVM语言都灵活得多。有关Java的异常引发规则以及`java.lang.RuntimeException`类在Java语言中扮演的重要角色，将在下一章讨论。

2.3.4 集合 API——`java.util.ArrayList` 和 `java.util.HashMap`

`java.util`包包含各种各样的数据结构，这里只介绍其中的两个，但后面会时不时地提及其他的。有些JVM语言使用这些类的变种，以提供额外的功能，但大多数JVM语言都使用这里讨论的类，以最大限度地与Java和JVM平台兼容。

很多语言都支持**泛型**（generics），以限制对象可使用的对象类型。在集合类中，通过使用泛型限制的是可在集合类中存储的对象类型。鉴于很多语言都有不同的泛型表示规则，这个主题将在讨论各种语言时阐述，这里就按下不表了。在本书介绍的语言中，Clojure当前根本就不支持泛型。

需要指出的是，集合类只能用于处理对象。对于基本数据类型值，将把它们自动装箱为对象，反之亦然；这也适用于支持基本类型值的其他JVM语言。自动装箱在前面介绍基本类型值时讨论过。

本节将介绍如下两个集合类：

- `java.util.ArrayList`（一个列表类，在内部是使用数组实现的）；
- `java.util.HashMap`（包含键–值组合的容器）。



在Python程序员看来，这两个类分别相当于列表和字典；而在Ruby程序员看来，它们分别相当于Ruby中的`Array`和`Hash`对象。

1. `ArrayList`（`java.util.ArrayList`）

这个类非常简单，使用起来也很方便。顾名思义，它实现了可存储其他对象的链表结构。

虽然JVM平台和大多数JVM语言都提供了内置的数组支持，但`ArrayList`对象使用起来更容易。相比于常规数组，`ArrayList`有如下两个优势。

- 在已经填满且需要更多空间时，`ArrayList`对象会自动加长，而数组必须手动进行管理。

- 数组只提供了一个属性（用于获取数组的长度），而ArrayList类提供了大量便利的方法。



虽然在JVM中，数组没有内置方法，但Java提供的java.util.Arrays类有很多方法，让数组使用起来更容易。有些JVM语言甚至给数组添加了额外的方法。

下面来看一些方法和代码示例。

(1) ArrayList类中常用的方法

下表列出了ArrayList对象提供的一些最重要的方法。请注意，使用泛型时，ArrayList对象处理的不是对象，而是指定的类型。

方 法 名	返回类型	描 述
add(Object o)	boolean	将指定对象添加到内部列表（通常是使用数组实现的）中
add(int index, Object o)	–	将指定对象添加到列表的指定位置
addAll(Collection c)	boolean	将指定集合中的所有项添加到当前列表中，其中 Collection 是一个接口，集合 API 中的很多类都实现了它
clear()	–	清除所有的内容
contains(Object o)	Object	指出指定的对象是否包含在列表中
get(int index)	Object	获取指定索引处的对象
set (int index, Object o)	Object	将指定索引处的对象替换为传入的对象
size()	int	返回列表包含的元素个数

(2) ArrayList使用示例

下面是一个简单的Java语言示例，演示了ArrayList的一些方法：

```
ArrayList list1 = new ArrayList();
list1.add("this is a test");
list1.add(0, "Hello");

ArrayList list2 = new ArrayList();
list2.addAll(list1);
list1.clear();

System.out.println(list1);
System.out.println(list2);
System.out.println(list2.contains("this is a test"));
```

输出如下：

```
[]
[Hello, this is a test]
true
```

第一行输出[]表明list1为空，而第二行输出表明list2包含两个字符串，依次为Hello和

this is a test。最后，向控制台打印了单词true，因为this is a test包含在ArrayList对象list2中。

2. HashMap（java.util.HashMap）

HashMap存储键–值组合。在JVM中，这种数据结构被称为映射。在映射中插入对象时，需要同时指定键对象和值对象；有了键对象，就可获取与之相关联的值。这种数据结构不保留键的插入顺序。

从技术上说，HashMap计算键对象的散列值，并以能够快速查找的方式存储它们，再存储与键相关联的值。下面先来看一些常用方法和代码示例，再更详细地介绍这个类的工作原理。

(1) HashMap类中常用的方法

HashMap类提供了很多方法，下表列出了其中最常用的，但要正而八经地使用HashMap类，务必参阅完整的API文档。

方 法 名	返回类型	描 述
put (Object key, Object value)	Object	添加指定的键–值对。如果指定的键已经存在，则使用指定的值替换原来与之相关联的值。如果添加了指定的键，就返回 null，否则返回原来的值
putAll (Map map)	–	添加指定映射中所有的键–值对，同样，对于已存在的键，替换与之相关联的值
putIfAbsent (Object key, Object value)	Object	仅当指定的键不存在时，才添加指定的键–值对。如果指定的键已存在，就什么都不做。在添加了指定的键–值对时返回 null；如果指定的键已存在，就返回原来与之相关联的值
remove (Object key)	Object	如果指定的键已存在，就删除相应的键–值对，否则什么都不做
containsKey (Object key)	boolean	指出指定的键当前是否包含在映射中
get (Object key)	Object	返回与指定的键相关联的值；如果没有找到指定的键，就返回 null
getOrDefault (Object key, Object defaultValue)	Object	如果找到指定的键，就返回与之相关联的值，否则返回 defaultValue
clear()	–	清空集合，即删除所有的键–值对
size()	int	返回映射当前存储的键–值对个数

(2) HashMap使用示例

下面的Java代码演示了HashMap的一些基本用法：

```
HashMap map = new HashMap();

map.put("key1", "value1");
map.put("key1", "value2");
map.putIfAbsent("key1", "value3");
```

```
System.out.println(map.get("key1"));
System.out.println(map.containsKey("value2"));
System.out.println(map.size());
```

输出如下：

```
value2
false
1
```

第一行输出为`value2`，因为第二次调用`map.put`时将原来的`value1`替换成了`value2`，而方法调用`map.putIfAbsent`什么都没做。第二行输出为`false`，因为方法`map.containsKey`只查找键。最后一行输出为`1`，因为只存储了一个键–值对。

3. 让自定义类的对象能够存储在集合API中

前面说过，基类`java.lang.Object`包含如下两个重要的方法：

- `hashCode()`
- `equals(Object other)`

所有集合API都大量地使用了这两个方法。为提高性能，同时确保集合API能够像预期的那样工作，在要存储到集合对象中的类中，必须重写这两个方法以提供良好的实现。创建自定义类时，Java程序员必须自己编写这两个方法的实现；但本书介绍的其他语言都通常会在你定义类时自动为这两个方法生成实现。



如果你不喜欢JVM语言替你生成的方法`hashCode()`和`equals()`的实现，在大多数情况下都可手动重写这些方法，以便提供自己的实现。

鉴于需要提供这两个方法的实现的大多是Java程序员，有关这两个方法需要遵循的规则将在第3章介绍。然而，这里将简要地介绍这些方法，让你知道众多的集合类是如何使用散列机制的。

(1) `hashCode()`简介

顾名思义，这个方法在需要获得当前对象的散列值时被调用。

方法`hashCode()`必须返回一个随对象内容变化而变化的整数值。另外，在对象类似的情况下，它应尽可能返回不同的值。



返回不能标识对象的值不算错，但这将给大多数集合类的性能带来负面影响。这一点将在后面更详细地讨论。

(2) `equals()`简介

方法`equals()`在传入的对象与当前对象相等时返回`true`，否则返回`false`。下面是一个简

单的示例：

```
Integer i = 25;
Object o = new Object();
System.out.println(i.equals(o));
```

上述代码将在控制台中打印false。

方法equals()必须检查两个对象，并指出它们是否类似。这个方法必须遵循很多规则，这也将在第3章讨论，因为通常只有Java开发人员才需要自己编写这个方法。



如果类提供的方法equals()的实现没有遵守所有的规则（也叫约定，contract），将无法保证集合类能够正确地工作。

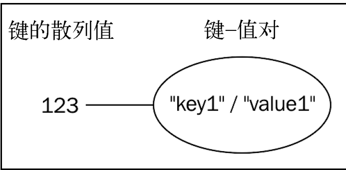
(3) 散列机制

为说明方法hashCode()和equals()为何如此重要，我们来看一个示例。

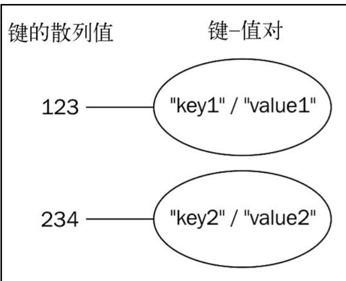
请看下面的Java代码，它创建一个HashMap实例并在其中添加一个键–值对：

```
map = HashMap();
map.put("key1", "value1");
```

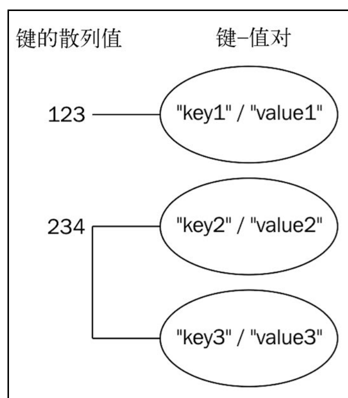
将对传入的键对象（这里是String实例key1）调用方法hashCode()。这将生成一个可用于散列的数字（在这个示例中，为123）。在内部，HashMap实例以能够快速查找的方式存储键的散列值，并将键和值都与之关联起来。



现在添加一个新的键–值对key2和value2。假设对对象key2调用方法hashCode()返回的是234，由于这个散列值未被占用，因此将添加它，并将指定的键对象和值对象与之关联起来。



接下来，添加新的键–值对key3和value3，但发生了意外：对String实例key3调用方法hashCode()时，返回的也是234，因此这个散列值同时指向键–值对key2/value2和key3/value3。这被称为**冲突**（collision）。



程序要求获取与键对象key3相关联的值对象：

```
Object o = map.get("key3");
```

HashMap对象将对传入的键"key3"调用方法hashCode()，这也将返回234。然而，HashMap对象发现这个散列值与两个键–值对相关联，因此对String对象"key2"和"key3"都调用方法equals()，以确定哪个键与String对象"key3"匹配。在这里，匹配的是"key3"，因此返回对象"value3"。

从这个示例可知，方法hashCode()和equals()非常重要。添加键-值对时，发生的冲突越少，找到键的速度越快。如果方法equals()的实现有问题，键查找过程也将失败。

2.4　从命令行运行 JVM 应用程序

在JVM中运行应用程序通常被视为一个极度复杂的主题。前一章说过，JVM编译器将源代码编译成扩展名为.class的二进制文件。要让JVM实例能够运行.class文件中的代码，必须遵循一些规则：

- 至少有一个类包含静态方法main()；
- 所有的类文件都必须存储在特定的目录中；
- 必须指定ClassPath；
- 类文件可放在JAR归档容器中；
- 要运行程序，可使用命令java。

我们将简要地介绍每条规则，然后使用Java编写一个演示项目，以尽可能清晰地说明这些规则。

2.4.1 至少有一个类包含静态方法 main()

使用命令java手动运行JVM应用程序时，需要指定包含如下静态方法的类：

```
public static void main(String[] args) {
}
```

JVM实例初始化完毕后，命令java将调用这个方法，它相当于C和C++中著名的入口函数main()，其中的字符串数组args包含操作系统传递的命令行参数。

上述代码片段演示的是Java的情况。有些JVM语言不要求程序员手工编写这个方法，而会在编译类时自动生成它。另外，有些JVM框架会自动生成或提供这个方法。

方法main()并非必须与前面演示的完全相同。

- 这个方法的参数的名称无关紧要。根据约定，将其命名为args，有时使用名称argv，但使用其他类似的名称也可行。
- 参数的类型是可以调整的，通常为字符串数组（在Java中为String[]），但使用数量可变的参数（String...，这将在下一章讨论）也可行。

下面是一个有效的Java main()函数，其中参数类型是String...，参数名称也不是args：

```
public static void main(String... commandLineArguments) { }
```

在项目中，可以有多个类包含静态方法main()，但每次只能运行其中的一个。使用命令java运行应用程序时，必须在命令行中指定全限定类名。

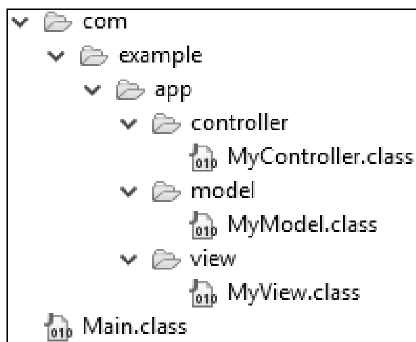
2.4.2 存储类文件的目录结构

类文件必须存储在与包名匹配的目录结构中。包名中的每个句点都意味着一个新的子目录。没有放在包中的类存储在根目录中，这个目录包含各个子目录。

假设有一个项目，它包含的类的全限定名如下：

- Main
- com.example.app.model.MyModel
- com.example.app.view.MyView
- com.example.app.controller.MyController

则编译后，目录结构将类似于下面这样：



在项目中，应避免包含未放在包中的类。在前面的示例中，最好将`Main`类放在一个包中。

请注意，编译器通常会创建正确的目录结构。为更好地理解ClassPath，务必熟悉这里介绍的需求。另外，有些编译器（包括Java编译器）要求以同样的方式组织源代码。

2.4.3　为 JVM 实例设置 ClassPath

ClassPath是一个由目录和/或JAR归档文件组成的列表，JVM将使用它来查找项目中引用的类。命令`java`、编译器（Java和其他大多数JVM语言编译器）以及其他与JDK和JVM相关的众多工具都要用到它。

如果没有显式地设置，ClassPath将默认为当前目录。如果项目使用的所有类文件都存储在用来启动程序的目录中，且类文件的包名与目录结构匹配，就无需显式地设置ClassPath。

在现实世界中，通常会用到附加库。前一章说过，很多JVM语言都要求加载运行时库；如果没有它，应用程序将无法运行。通常，将这种附加库文件（常被称为**依赖项**）放在一个独立的子目录中。库放在JAR文件中时（通常如此），你只能在ClassPath指定它，因为JVM不会自动加载JAR文件。

下面来看一个真实的示例，这是在我的Windows计算机中启动开源应用程序服务器Apache TomCat所需的ClassPath：

```
C:\apache-tomcat-8.0.44\bin\bootstrap.jar;C:\apachetomcat-
8.0.44\bin\tomcat-juli.jar
```

显然，启动Apache TomCat所需的类存储在两个JAR文件中：bootstrap.jar和tomcat-juli.jar。这些文件存储在Apache Tomcat安装目录下的子目录bin中。JAR文件包含一系列的类文件，这将在后面更详细地讨论。

对于目录和JAR文件，可指定其绝对路径，也可指定其相对路径。相对路径的起点为执行命令（如`java`或`javac`）时所在的目录。按从左到右的顺序读取ClassPath中的条目，直到找到指定

的类，因此条目的排列顺序至关重要。

可在多个层级设置ClassPath。JVM按如下顺序选择要使用的ClassPath。

- 如果设置了环境变量CLASSPATH，就使用该环境变量。
- 如果在执行命令java（或其他JDK工具，如Java编译器javac）时指定了命令行选项-cp或-classpath，就使用该选项。其他工具可能要求指定不同的命令行选项。



设置环境变量CLASSPATH的做法真心不推荐，因为需要同时运行多个JVM应用程序时，将很难管理这个环境变量。

主要的JVM应用程序大都包含自动设置ClassPath的简单的操作系统shell脚本（用于Linux/macOS中）和批处理文件（用于Windows中），让用户只需启动脚本就能启动程序。有些应用程序甚至包含原生的可执行文件，它设置正确的ClassPath并在幕后启动JVM，从而对用户隐藏应用程序使用了JVM的事实。

为让开发人员能够在测试或开发期间轻松地启动JVM应用程序，大多数构建工具都提供了相应的任务（或实现这个任务的插件），让开发人员只需执行一个命令就能自动设置ClassPath并启动应用程序。第4章将介绍流行的构建工具Gradle提供的这种便利功能。

2.4.4　将类文件放在 JAR 归档文件中

为方便起见，可将一系列类文件归档为单个JAR文件。JAR文件是标准的ZIP文件（只是文件扩展名不同），但不同于ZIP文件，JAR文件有严格的内容管理规则。在ClassPath中指定JAR文件时，将加载其中所有的类，让JVM实例能够使用它们。

本章不会讨论如何创建JAR归档文件，而将这个主题留到第4章构建Java语言示例时再介绍。



JAR文件可能有其外部依赖，在这种情况下，必须在ClassPath中包含这些依赖项。在提供JAR文件的库或工具的文档中，通常会指出这些外部依赖。

可运行的JAR文件

可对JAR文件进行设置，以便能够使用命令java来运行它，但仅当正确地配置了JAR文件时才有可能。在这种情况下，JAR文件指定了哪个类包含将被命令java运行的方法main()。

对最终用户来说，这很方便，因为JAR文件是完全独立的：

- JAR文件包含所有必要的依赖项；
- 根本不需要（甚至无法）手动设置ClassPath；
- 用户无需告诉JVM哪个类包含方法main()。



存在一些局限性：可运行的JAR文件无法查找它不包含的类；可运行的JAR文件中的类不能使用ClassPath。

2.4.5　使用命令 java 运行程序

命令java用于启动JVM实例和应用程序。面临的情形有两种：

- 运行由独立的类文件组成的项目；
- 运行存储在可运行的JAR文件中的项目。

我们还将介绍命令java的一些重要参数。

1. 运行由独立的类文件组成的项目

项目存储在包含类文件的目录中时，通常像下面这样使用命令java来运行它（即便项目依赖于JAR文件亦如此）：

```
java -cp "CLASSPATH" MAINCLASS ARGUMENTS
```

你需要将CLASSPATH替换为实际使用的类路径，并将MAINCLASS替换为包含静态方法main()的类的全限定名。如果这个类支持参数，可将ARGUMENTS替换为所需的参数。

下面来看一个真实的示例。这个示例摘自Oracle JDK组件JavaDB提供的Windows 批处理脚本，但稍微做了简化。从JDK安装目录的子目录db执行这个命令时，它将启动Apache Derby Network Server：

```
java -cp
"lib\derby.jar;lib\derbynet.jar;lib\derbyclient.jar;lib\derbytools.jar;lib\
derbyoptionaltools.jar" org.apache.derby.drda.NetworkServerControl start
```

这个示例清楚地表明，必须在类路径中逐个指定所有必要的JAR文件，因为JVM实例根本不会尝试去加载未在类路径中指定的JAR文件。包含函数static void main()的类的全限定名为org.apache.derby.drda.NetworkServerControl，而向这个函数传递的命令行参数为start。由于每个类路径条目指定的都是一个JAR文件而不是目录，因此指定的类必须包含在其中的一个JAR文件中。

在较新的JRE版本中，可指定通配符，这将加载匹配的JAR文件。例如，前面的示例可简化成下面这样：

```
java -cp "lib\*" org.apache.derby.drda.NetworkServerControl start
```

为加载JAR文件，通配符*必不可少。如果只指定目录名，而没有通配符，该目录中的JAR文件将不会添加到ClassPath中，而只会添加目录中的.class文件。

2. 运行放在可运行的JAR文件中的项目

要自动运行经过正确配置的JAR文件（这在前面讨论过，并非所有的JAR文件都如此），可像下面这样使用命令java：

```
java -jar PATH
```

请将PATH替换为JAR文件的绝对路径或相对路径。如果JAR文件配置正确，这将运行程序。

请注意，在这种情况下，无法设置类路径（因为将忽略环境变量CLASSPATH以及命令java的-cp和-classpath参数），因此指定的JAR归档文件必须包含所有必要的依赖项。

3. 命令java的其他很有用的参数

要查看完整的选项列表，可执行命令java而无需指定任何选项。

其中一些需要注意的选项如下：

- 传递属性和值的选项-D；
- 启用断言的选项-ea。

有些选项既有简写形式，又有冗长形式；但这里只列出了简写形式。另外，参数是区分大小写的。

(1) 传递属性和值的选项-D

-D用于设置属性。属性是可在代码中读取的字符串，可以多种方式传递给JVM，包括使用选项-D。可使用这个选项多次：要传递给程序的每个参数/值对一次。

下面是一个这样的示例：

```
java -cp CLASSPATH -DProperty1=Value1 -DProperty2=Value2 MAINCLASS
```

要在代码中读取属性，可使用java.lang.System类的方法getProperty。在后面的示例中，还使用了这个方法来读取预定义的系统属性。

(2) 启用断言的-ea

使用这个选项可启用默认被关闭的断言。

在支持断言的语言中，程序员可添加运行阶段条件检查。在Java中，这是通过添加assert语句并在其中指定条件实现的。断言被禁用时，这些语句将被忽略，但断言被启用时，JVM将在条件不满足时引发错误。这可用来检查程序是否像期望的那样工作。下面是一个Java assert语句：

```
int i = 25;

assert i < 24;
```

使用选项-ea启用了断言时，上述assert语句将导致JVM引发java.lang.Error。如果没有在命令行中显式地指定选项-ea，上述代码将不会引发错误。

可全局启用断言，也可针对特定的包启用断言，为此可使用-ea:PACKAGE，其中PACKAGE必须替换为完整的包名。你可为每个要启用断言的包指定选项-ea:PACKAGE。

要详尽地测试代码，编写单元测试是更佳的做法，但断言在有些情况下也能派上用场。

2.4.6　在 JVM 中运行的示例项目

下面来创建一个稍微有点设计过度的程序，它在控制台中打印一些JVM信息，由三个类组成。这里不会使用IDE，而使用常规文本编辑器来编写代码，并使用命令提示符（Windows）或Terminal（macOS/Linux）来编译代码。最后，我们将从命令行运行这个应用程序。这个项目中的类放在下面几个包中：

- com.example.app
- com.example.app.model
- com.example.app.view

请创建一个根目录来存储源代码文件和编译得到的文件，再在这个目录中创建子目录src和bin。

在目录src中，创建如下子目录：

- com
- com\example
- com\example\app
- com\example\app\model
- com\example\app\view

启动你喜欢的文本编辑器，并在子目录model中创建一个名为ModelFoo.java的文件，它包含如下内容：

```
package com.example.app.model;

public class ModelFoo {
  public String getJVMInfo() {
    return "JVM version " + System.getProperty("java.version") +
    " by " + System.getProperty("java.vendor");
  }
}
```

ModelFoo类只包含一个公有方法，这个方法返回一个String对象，其中包含一些有关当前使用的JVM的信息。在Java中，可直接使用System类，这将在第3章介绍。这个类的静态方法

getProperty()返回指定属性的值，这里使用的两个属性都是内置的，它们分别包含JRE版本和厂商。

在子目录view中，创建文件ViewBar.java：

```
package com.example.app.view;
import com.example.app.model.ModelFoo;

public class ViewBar {
  public void showJVMInfo(ModelFoo model) {
    System.out.println("This program is running on " +
                       model.getJVMInfo());
  }
}
```

这个类在控制台中打印对象model提供的版本信息。

最后，在子目录app中创建文件Controller.java：

```
package com.example.app;
import com.example.app.model.ModelFoo;
import com.example.app.view.ViewBar;

public class Controller {
  public static void main(String[] args) {
    ViewBar view = new ViewBar();
    view.showJVMInfo(new ModelFoo());
  }
}
```

Controller类将其他两个类粘合起来，且包含方法main()。这个示例没有很好地实现"模型–视图–控制器"设计模式，因为为节省篇幅，我有点偷工减料了。

请注意，子目录src的结构与包名匹配。这是Java遵循的一个约定，其他一些语言并未遵循这种源代码文件存储约定，但JVM要求编译后的文件必须遵循这种存储约定。

打开命令提示符（Windows）或Terminal窗口（macOS/Linux），并切换到这个项目的根目录（即子目录src和bin所在的目录）。运行下面的命令来编译代码（请按你使用的操作系统要求的方式指定Controller.java的路径，这里遵循的是Windows使用的约定）：

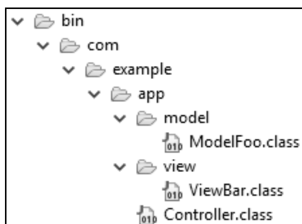
```
javac -sourcepath src -d bin src\com\example\app\Controller.java
```

这里发生了很多事情。

- 选项-sourcpath src告诉编译器，所有的源代码都位于子目录src中。
- 选项-d bin让javac将编译得到的文件放在子目录bin中。这个目录必须存在，但javac会根据需要自动创建子目录。
- 最后，传递了主程序的源代码文件的路径。

由于源代码文件Controller.java导入了其他两个类，且目录src的结构与所有的包名都匹配，因此Java编译器能够找到所有的类并编译它们。

这将生成如下图所示的输出目录bin：



下面来使用命令`java`运行这个应用程序。为此，在命令行窗口中切换到子目录bin，再执行如下命令：

```
java com.example.app.Controller
```

在我的计算机上，输出如下：

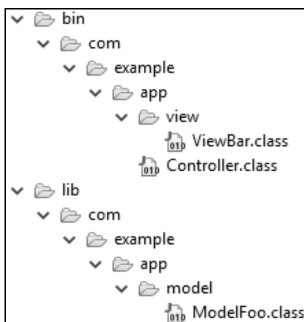
```
This is running on JVM version 1.8.0_112 by Oracle Corporation
```

ClassPath示例

为演示ClassPath的工作原理，我们将一个类文件移动另一个目录中。从概念上说，这与使用外部依赖没什么不同，因为根据约定，外部依赖也与项目的类存储在不同的目录中。请执行如下步骤：

- 在项目的根目录（即包含子目录src和bin的目录）中，创建目录lib；
- 在新创建的目录lib中，创建子目录com\example\app\model；
- 将文件ModelFoo.class移到刚创建的子目录model中；
- 为清晰起见，将空目录bin\com\example\app\model删除。

现在，目录结构应类似于下图这样：



在命令提示符或Terminal窗口中，切换到项目根目录下的子目录bin，并尝试再次运行这个程序：

```
java com.example.app.Controller
```

你将看到一个Java栈跟踪。请习惯这种情况，因为在JVM开发过程中，你经常会遇到这样的错误。在我的计算机上，出现的栈跟踪类似于下面这样（为简洁起见有删节）：

```
Error: A JNI error has occurred, please check your installation and try
Again
Exception in thread "main" java.lang.NoClassDefFoundError:
com/example/app/model/ModelFoo
 at java.lang.Class.getDeclaredMethods0(Native Method)
 at java.lang.Class.privateGetDeclaredMethods(Unknown Source)
 at java.lang.Class.privateGetMethodRecursive(Unknown Source)
 at java.lang.Class.getMethod0(Unknown Source)
...
```

现在让`java`命令在当前目录和lib目录（它位于当前目录bin的父目录中）中查找代码中引用的类，为此可使用选项`-cp`来设置ClassPath：

```
java -cp ".;..\lib" com.example.app.Controller
```

查找类时，JVM将首先在当前目录中查找，如果找不到，再在子目录..lib中查找。请注意，必须在类名前指定选项`-cp`及其值，否则它们将被传递给main函数的String数组参数，而不是命令`java`。

2.5 Eclipse IDE

正如你在前一节看到的，使用简单的文本编辑器来创建JVM程序相当繁琐。在包括Java在内的有些语言中，你必须确保包名与源代码的目录结构匹配。稍后你将看到，有些语言还要求开发人员遵守其他规则，例如，Java要求源代码文件名与相应的类名匹配。另外，运行程序时，你必须手工指定ClassPath。类似这样的要求还有很多。

在JVM领域，程序员大多使用精致的IDE来开发项目。在市面上，有支持JVM的商业IDE，也有开源的IDE。所有流行的IDE都提供了强大的Java支持，而现代IDE都提供了如下功能。

- 首先是自动补全功能。识别类名后，IDE将显示其成员列表（Microsoft称之为智能感知）。
- 其次，IDE提供了尖端的重构工具。当你重命名变量或方法时，将自动修改整个项目的代码，以反映你所做的修改。
- 提供了功能齐备的GUI调试器，它们支持断点、变量检查和性能分析。
- 提供了自动重写既有代码以使用新的Java功能的选项。
- 就Java编译器未发现的问题提出警告，如访问空引用的成员。

- 你只需单击一个按钮就能运行项目本身或项目的单元测试。
- 提供了自动将Java EE项目部署到JVM应用程序服务器的功能。
- 还提供了其他工具，如对话框生成器、可视化的数据库工具（SQL）等。
- 支持使用插件来添加其他功能。

正如你将在本书后面看到的，在某些方面，IDE对除Java之外的其他JVM语言的支持还有不足之处，但相比于几年前，已经有很大的改进了。

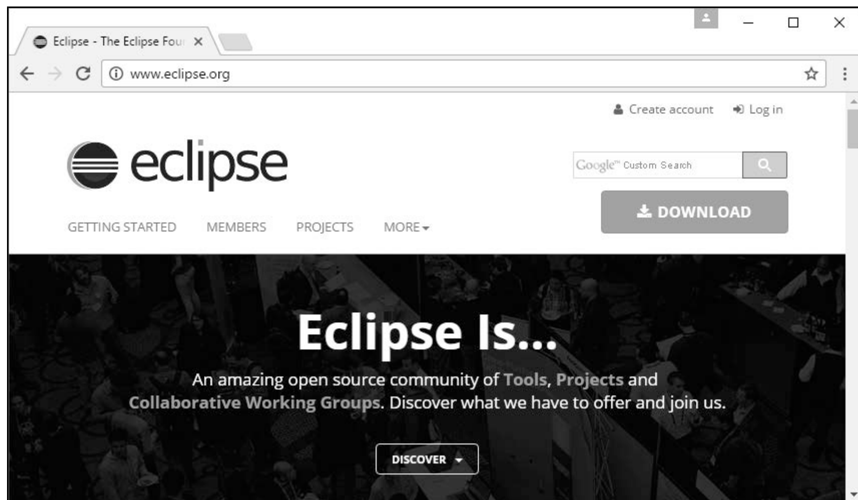
下面是JVM开发人员可使用的最著名的IDE。

- IntelliJ IDEA（一款现代IDE，有功能齐备的商业版，也有更简单但免费的社区版）。
- Apache NetBeans IDE（其前身为Oracle NetBeans，这个IDE以强大的构建工具支持和大量的内置功能著称，它还支持插件）。
- Eclipse IDE（来自成员包括IBM和其他大型公司的Eclipse Foundation的杰出产品；与NetBeans IDE一样，也可使用插件对其进行扩展）。

NetBeans IDE和Eclipse IDE都是开源项目，而IntelliJ是专用的。在本书中，我选择的是Eclipse IDE，这是一个艰难的抉择，因为前述IDE都非常出色，且都有缺点和优点。对于本书介绍的语言，Eclipse IDE提供的支持是最好的，虽然使用它必须安装一些外部插件。

2.5.1　下载 Eclipse IDE

要下载Eclipse IDE，可访问http://www.eclipse.org，再单击Download按钮，如下图所示。



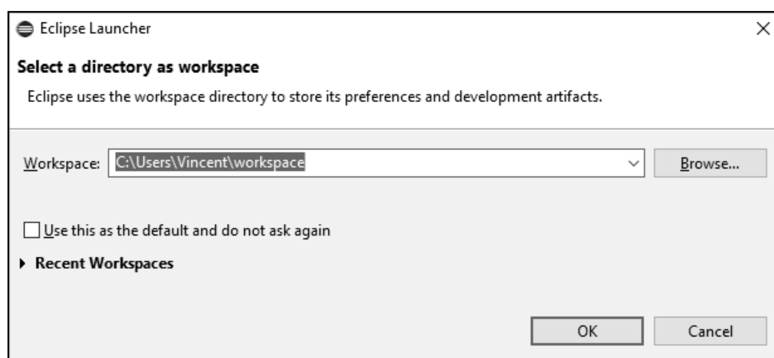
当前，对于所有平台，Eclipse IDE都使用对用户友好的安装程序。

2.5.2 安装 Eclipse IDE

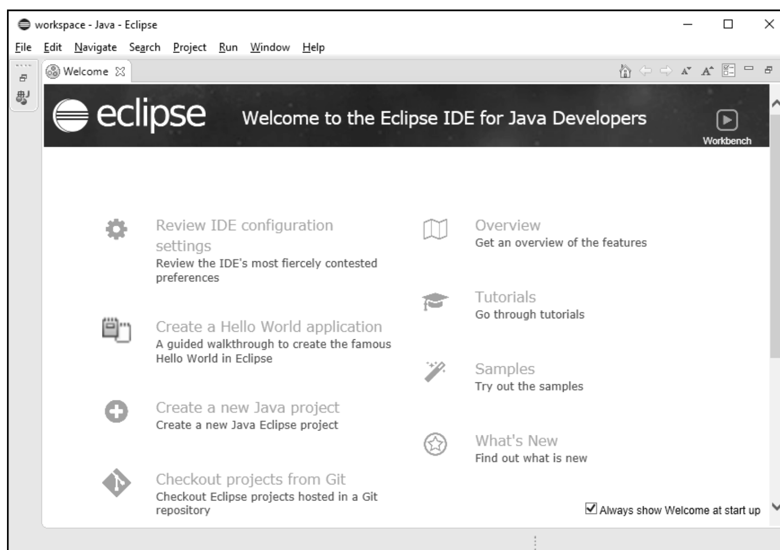
Eclipse IDE安装起来非常简单，较新的版本都可使用GUI安装程序来安装：

- 启动下载的安装程序；
- 在安装程序询问时选择Eclipse IDE for Java Developers；
- 选择安装目录再单击Install。

安装完毕后，通过运行来检查是否正确地安装了程序。首先将显示一个开始屏幕，过段时间后，将出现一个窗口，让你指定工作区目录（即存储项目的目录），如下图所示。



接受默认设置并单击OK按钮，将出现一个欢迎屏幕，如下图所示。



在本书后面，我们将提供有关如何安装所需插件的说明。

2.6　小结

本章介绍了很多主题，下面来全面地回顾一下。

你下载并安装了JDK，然后对其进行了详细探索：查看其目录结构并研究其最重要的命令。你学习了如何将类组织成包，这种知识在你研究Java类库时很有用。你学习了`java.lang`包中一些重要的类，还研究了`java.util`包中的重要类`ArrayList`和`HashMap`。你编写、编译并在JVM实例中运行了一个简单的程序，还学习了如何修改ClassPath。最后，为提高效率，你下载并安装了Eclipse IDE。

祝贺你！牢固地掌握JVM概念后，就可深入探索JVM编程语言了。我们先来看第一种JVM编程语言——Java。

第 3 章

Java

前两章探索JVM和JDK时，我们列举了很多Java代码。使用Java编写的源代码通常易于阅读和理解。Java刚推出时是一种学习起来比较简单的语言，多年来随着添加的功能越来越多，其复杂程度有所上升，但好消息是，在没有做好准备的情况下，初学者无需过多关心较高级的主题。

即便你选择使用其他JVM语言，也可受惠于本章，尤其是你开始使用根据Javadoc注释生成API文档的库或框架时。稍后你将看到，Javadoc是JDK提供的一个工具，用于根据源代码中的特殊注释来生成HTML文档。很多库和框架都在其文档中包含Javadoc生成的HTML文档。

本章讨论如下主题：

- Java的面向对象编程（OOP）功能；
- 使用Java进行编程。

3.1 Java 中的面向对象编程功能

前两章说过，在Java中，除基本类型外的其他一切都是对象。Java支持基本类型，因此不是纯粹的OOP语言，但不失为一种严肃的OOP语言。

要有效地使用Java，你必须熟悉OOP。如果你将OOP忘得差不多了，也不用担心，因为本章将力图帮助你复习这方面的知识，虽然不会全面介绍OOP。本章重点介绍各种与OOP相关的主题：

- 定义类；
- 定义包；
- 添加类成员——变量和方法；
- 构造函数和析构函数；
- 继承；
- 接口；
- 抽象类；
- 向上转换和向下转换。

3.1.1　定义类

从前两章的示例可知，要定义类，只需使用Java关键字`class`，并在它后面加上类名和大括号（`{ }`）。其中大括号让程序员知道类包含哪些代码：

```
class ClassName {
}
```

上述代码遵守了所有的Java语法规则，因此能够通过编译，但删除其中的任何一部分都将导致编译错误。

给JVM类命名时，应采用骆驼拼写法：类名的第一个字母大写；如果类名由多个单词组成，单词之间不用空格或下划线分隔，且每个单词的首字母都大写。给类命名时需遵守如下规则。

- 类名不能以数字打头。
- 类名不能包含连字符或空格；类名可包含下划线，但根据约定不使用它。
- 类名可包含数字，但不能以数字打头。
- 类名不能是Java保留的关键字。要将关键字用作类名，必须至少修改或添加一个字符，这样才能避免类名违反这种规则。

3.1.2　类访问限定符

类的可见性是可调整的。在没有明确指定的情况下（就像前面的示例那样），可见性是包私有的，即只能在所属的包中引用和实例化它。有关JVM中包的工作原理的详细说明，请参阅前一章。

在大多数情况下，你都希望创建可在任何地方引用和实例化的类。为此可在关键字`class`前面添加访问限定符`public`，这样在任何包中都能够看到并实例化当前类：

```
public class ClassName {
}
```



即便是公有类，也可包含私有的构造函数。这样的类可在其他包中看到并引用，但不能访问其构造函数的代码无法实例化它。

Java编程语言有一个不同寻常的要求，那就是在一个源代码文件中，只能定义一个公有类，且文件名必须与类名完全相同。其他JVM语言通常没有这样的限制。

3.1.3　类限定符 **`final`**——锁定类

可在类名前添加非访问限定符`final`，这将禁止其他类继承当前类：

```
public final class ThisClassCanNotBeOverriden {
}
```

关键字final可放在访问限定符的前面，也可放在它后面，但根据约定，先指定访问限定符，再指定非访问限定符。如果你试图继承final类，编译器将拒绝对代码进行编译。

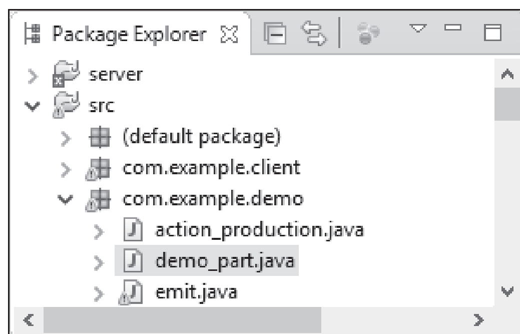
3.1.4 定义包

要将类放在包中，可使用关键字package。指定类所属包的代码必须是第一行非注释代码；与其他Java语句一样，这种语句也以分号（;）结尾，如下所示：

```
package com.example.package_name;
```

前一章详细讨论过包，包括命名约定和需求。未指定所属的包时，类将被加入到默认包中。

根据约定，存储Java源代码文件的目录的结构必须与包名匹配。所有流行的IDE都能看懂包结构，它们不显示各个子目录，而是显示完整的包名。例如，下面是Eclipse IDE的项目资源管理器（project explorer）的屏幕截图：



3.1.5 导入类

要在代码中引用类，可使用全限定类名。例如，要在方法中使用ArrayList，可像下面这样编写代码：

```
java.util.ArrayList list = new java.util.ArrayList();
```

这要求大量的键击，即便对以繁琐著称的语言来说亦如此。当然，Java提供了解决方案：通过使用关键字import，引用类时只需使用其名称。import语句的最基本形式类似于下面这样：

```
import java.util.ArrayList;
class Demo {
  ArrayList list = new ArrayList();
}
```

在这个方法的代码中，现在只需使用类名`ArrayList`，而无需再使用全限定类名。`import`语句必须位于指定类所属包的语句后面，同时位于第一行类定义代码前面。在一条`import`语句中，不能指定多个包。



类名发生冲突（即在多个包中使用相同的类名）时，可只导入其中的一个类。但在这种情况下，要在代码中引用其他的同名类，必须使用全限定类名。

还可在一条`import`语句中导入指定包中的所有类：

```
import java.util.*;
```

然而，为较大的系统编写代码时，不推荐使用这种形式的`import`语句，因为这会增加出现类名冲突的风险。请注意，这只会导入指定包中的所有类，而子包不受影响，即必须单独导入它们。例如，`java.util.concurrent`包包含用于并发编程的实用类。要导入`java.util`和`java.util.concurrent`包中的类，必须分别使用不同的`import`语句。



默认导入了`java.lang`包中的所有类，你可直接使用它们。

3.1.6　添加类成员——变量和方法

如果没有变量和方法，类将既乏味又无用。变量用于存储通常由方法进行处理的数据。与类一样，变量和方法的可见性也是可以修改的，为此可在它们前面加上访问限定符，但它们还支持其他的限定符。我们将先讨论定义变量和方法的语法，再讨论限定符。因此本节包含如下主题：

- 实例变量；
- 实例方法；
- 访问限定符；
- 限定符`static`；
- 限定符`final`；
- 方法重载。

1. 实例变量

通常，每个对象实例都有自己的变量，这些变量被称为**实例变量**（instance variable）。在Java中，实例变量是像下面这样定义的：

```
TYPE variableName;
```

其中`TYPE`可以是任何基本类型（`int`、`double`等）和引用类型。如果你要使用的类已经使用`import`语句导入了，则只需指定类名，否则必须指定全限定类名。可在声明变量的同时对其

进行初始化：

```
public class Test {
  int i = 25;
  Object o = new Object();
}
```

对于未在类级显式地初始化的变量，其值将被隐式地初始化为0（如果其类型为int、long或short）、0.0（如果其类型为float或double）、false（如果其类型为boolean）或null（如果其类型为引用）。对变量名的要求与类名相同，但命名约定不同。根据约定，变量名以小写字母开头。与类名一样，变量名也可包含$和下划线等特殊字符，但不推荐这样做。

2. 方法

在Java中，函数可返回对象或null。对于什么都不返回的方法，必须使用关键字void，这借鉴了C语言。常规类中的方法必须包含放在大括号（{ }）内的方法体：

```
public class ClassWithTwoMethods {
  boolean b;
  void methodReturnsNothingAndNoParameters() {
  }

  Object methodReturnsAnObject(boolean b, int i) {
    this.b = b;
    return null;
  }
}
```

有返回类型的方法必须在方法体中返回相应的对象或null，否则将无法通过编译，但使用关键字void的方法什么都不返回。

在方法中声明的变量必须初始化后才能使用。不同于类变量和实例变量，对于在方法中声明的变量，不会自动对其进行初始化。



注意：从前面的示例可知，在方法中可使用关键字this来访问类成员。

3.1.7　限定符

在变量和方法前面都可指定限定符。限定符分两类：

- 访问限定符；
- 非访问限定符。

很多访问限定符和非访问限定符都可结合起来使用。你将看到，指定多个限定符时，它们的顺序不重要。根据约定，应先指定访问限定符，再指定非访问限定符，但这并非是不可违背的规则。

1. 使用访问限定符保护类成员

要指定类的哪些成员（变量和方法）可被其他类访问，可在它们前面加上访问限定符。相比于类本身，可用于类成员的访问限定符更多。下表列出了可用于类成员的所有访问访问限定符。

名　称	访问限定符	描　述
公有	`public`	公有成员可被能够访问其所属类的所有代码访问
受保护	`protected`	受保护的成员在当前类、当前包的其他类以及继承当前类的类中可见且可访问，但对其他所有的类来说都不可见
包私有		没有指定访问限定符时，类成员仅在当前类和当前包的其他类中可见且可访问，但对其他包中的类来说是不可见的，对继承了当前类的类来说亦如此
私有	`private`	私有成员仅在其所属的类中可见且可访问，在其他类中都不可见也无法访问

继承其他类的类可重写被继承类中它能够访问的所有方法，但使用限定符`final`定义的方法除外（这一点你马上就将看到）。在任何情况下都不能重写私有方法，因为类无法重写它看不到或不能访问的方法。



这是Python程序员不熟悉的地方。在Python中，类中的所有变量和方法都是公有的，因此可在任何地方修改类变量以及调用类的方法，即便它们是仅供内部使用的。

- 访问限定符举例

下面来看一段包含两个类的Java源代码。

第一个类包含4个变量，而每个变量都使用了不同的访问限定符（或根本没有指定限定符）：

```
package chapter02.access_modifiers.demonstration;
public class DemoVariables {
  public String publicVariable = "This is a public variable";
  protected String protectedVariable = "This is a protected variable";
  String packagePrivateVariable = "This is a package-private variable";
  private String privateVariable = "This is a private variable";
}
```

请注意，这个类放在`chapter02.access_modifiers.demonstration`包中。

第二个类位于`chapter02.access_modifiers`包中，它创建第一个类的实例：

```
package chapter02.access_modifiers;
import chapter02.access_modifiers.demonstration.DemoVariables;
public class AccessModifiersMain {
  public static void main(String[] args) {
    DemoVariables demo = new DemoVariables();
    System.out.println(demo.publicVariable);
  }
}
```

虽然这两个包的名称都以chapter02.access_modifiers打头，但在JVM看来，它们之间毫无关系。



在JVM看来，只要包名不完全相同，它们就是不同的包。

另外，注意AccessModifiersMain类（以下称之为主类）创建了DemoVariables类（以下称之为演示类）的一个实例，但主类没有继承演示类。因此，我们可得出如下结论。

- 主类可随便访问演示类的变量publicVariable。事实上，这是主类能够看到并访问的唯一一个演示类成员。
- 主类不能访问演示类的变量protectedVariable，因为这两个类位于不同的包中，且主类没有继承演示类。只要这两个条件有一个不同，主类就能访问这个变量。
- 仅当主类和演示类位于同一个包中时，主类才能访问演示类的变量packagePrivateVariable。鉴于主类和演示类位于不同的包中，因此主类看不到也无法访问这个成员。
- 只有演示类才能访问变量privateVariable，其他任何类都无法访问。

2. 限定符static——实例变量和类变量

通常，你将创建类的每个实例独有的变量，并添加使用这些数据的方法。在这种情况下，类的每个实例都有自己的变量值，因此修改变量只影响当前实例。要调用实例方法，必须通过类的实例，即经过初始化的引用类型变量。

JVM也支持类变量和类方法，这些成员无需通过类的实例就能使用，但它们可在类的所有实例之间共享。要定义类变量或类方法，必须在它们前面指定非访问限定符static：

```
public class StaticDemo {
  public static String staticVariable = "This is a static variable";
  public String instanceVariable = "This is a class instance variable";
}
```

静态变量被称为类变量，在没有类的实例时也能访问。静态方法被称为类方法，在没有指向类实例的引用变量时也可调用。下面来看一个示例。为此，我们创建一个类，这个类创建了前述类的两个实例，并通过这两个实例修改了前述类中两个变量的值：

```
public class StaticDemoMain {
  public static void main(String[] args) {
    StaticDemo demo1 = new StaticDemo();
    demo1.staticVariable = "Demo 1 static";
    demo1.instanceVariable = "Demo 1 instance";

    StaticDemo demo2 = new StaticDemo();
    demo2.staticVariable = "Demo 2 static";
    demo2.instanceVariable = "Demo 2 Instance";

    System.out.println(StaticDemo.staticVariable);
    System.out.println(demo1.instanceVariable);
```

```
    System.out.println(demo2.instanceVariable);
  }
}
```

如果你运行这个程序，将生成如下输出：

```
Demo 2 static
Demo 1 instance
Demo 2 instance
```

请注意，在前述代码中，通过引用变量（demo1或demo2）来访问staticVariable时，无法知道这是一个静态变量；但通过System.out.println(StaticDemo.staticVariable)来引用这个变量时，这一点显而易见，因为它是通过类名StaticDemo来访问这个静态变量的。



通过引用变量类访问静态成员被认为是糟糕的做法，因为这隐藏了访问的是静态成员这样的事实。如果通过类名来访问静态成员，就不会留下任何令人迷惑的地方。

静态方法只能访问其所属类的静态成员，而不能直接使用任何实例变量，也不能直接调用任何实例方法。

3. 限定符final——锁定类成员

要锁定类的方法或变量，可在它前面加上非访问限定符final。下面的Java示例演示了一个final静态int变量和一个final方法：

```
class FinalDemo1 {
  public final static int THIS_IS_A_CONSTANT_VALUE = 42;
  public final void thisMethodCanNotBeOverridden() { }
}
```

将关键字final用于方法时，意味着任何类都不能重写它，而不管给这个方法指定的访问限定符是什么。如果有类依然试图重写这个方法，编译器将拒绝编译这个类。

将关键字final用于变量时，意味着它的值是不能修改的，这相当于将变量变成了常量。根据约定，final变量应声明为静态的，且名称应为全部大写。请注意，虽然final变量不可修改，但如果它指向的对象是可修改的，依然可以修改该对象的内容，下面的示例演示了这一点：

```
import java.util.ArrayList;

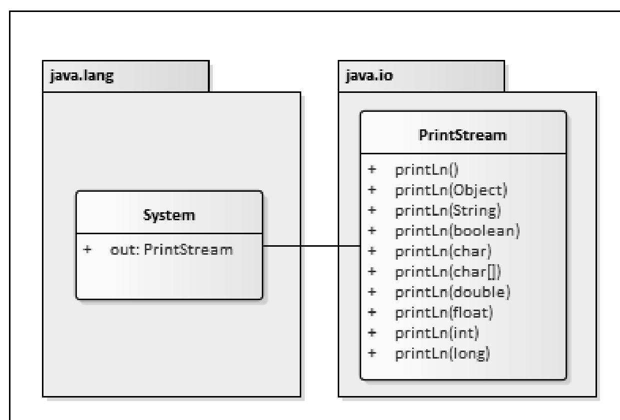
class FinalDemo2 {
  private static final ArrayList<String> finalList = new ArrayList<>();

  public static final void main(String[] args) {
    finalList.add("Both strings can be added, because");
    finalList.add("the ArrayList itself is mutable.");
  }
}
```

4. 重载方法

在Java中，可定义方法的不同版本。当你调用这种方法时，编译器将寻找匹配的版本。如果找不到完全匹配的版本，编译器将检查是否有能够接受指定参数的版本。如果有多个版本与指定的参数完全匹配，或者没有任何版本匹配，编译器将报错；否则，编译器将调用匹配的版本。我们先来看一个真实的示例——方法`java.lang.System.out.println()`，再介绍方法重载规则。

3



本书前面多次使用了方法`System.out.println()`，它来自`java.lang.System`类，而`System`类位于唯一一个Java默认自动导入的包`java.lang`中。`System`类包含公有静态变量`out`，这是一个只读的`final`变量，指向一个`java.io.PrintStream`对象实例。`println`是`java.io.Printstream`类的方法之一，它有多个重载版本，如前面的类图所示。

重载规则如下。

- 每个版本的参数类型和/或顺序必须不同。例如，如果两个版本都只接受一个`long`参数，编译器将无法确定该调用哪个。
- 理想情况下，同一个方法的所有重载版本的返回类型都应相同。如果两个版本只是返回类型不同，而参数类型和顺序相同，编译器将报错。
- 参数名一点都不重要。
- 如果没有找到完全匹配的版本，将拓宽基本类型值。鉴于`int`总是可存储在`long`变量中，因此认为接受`long`参数的方法与之匹配。反过来则不正确，因为在不丢失数据的情况下，不会总是能够将`long`值转换为`int`，因此不会尝试这样的转换。
- 如果没有完全匹配的版本，且至少有一个参数为基本类型，将把它自动装箱为包装类，再尝试与每个版本匹配。每次尝试对一个参数做这样的处理，直到找到匹配的版本或执行完所有这样的尝试。

- 这种处理也适用于为基本类型包装类的参数——将它们自动拆箱为基本类型。
- 对于每个为类实例的参数，将其转换为父类（或其实现的接口），并尝试与方法的每个重载版本匹配，直到找到完全匹配的版本或转换成了`java.lang.Object`类（本书前面说过，在JVM中，所有类都是从`Object`派生出来的）。

3.1.8　构造函数和终结方法

在Java中，可给类定义构造函数和终结方法（finalizer）：

- 构造函数在创建类的实例时被调用；
- 方法`finalize()`在对象被垃圾收集器收集时被JVM调用。

1. 构造函数

要定义构造函数，必须将其命名为与类同名，再加上用于包含参数的括号（`()`），如下所示：

```
public class ClassWithConstructor {
  public ClassWithConstructor() {
  }

  public ClassWithConstructor(int a, int b) {
  }
}
```

对于构造函数，可使用的访问限定符与普通方法相同。与普通方法一样，构造函数也可重载。实例化前面的类时，可使用下面两个构造函数中的任何一个：

```
ClassWithConstructor c1 = new ClassWithConstructor();
ClassWithConstructor c2 = new ClassWithConstructor(1, 2);
```

如果类没有定义任何构造函数，Java将隐式地生成一个。这样的构造函数类似于下面这样：

```
class ClassWithoutConstructor {
  public ClassWithoutConstructor() { }
}
```

前面说过，可在构造函数前面指定访问限定符。用于构造函数时，访问限定符的含义与用于普通方法时完全相同。与普通方法一样，如果没有指定任何访问限定符，构造函数将被视为是包私有的，这种构造函数只能被当前类所属包中的类调用。

2. 终结方法（finalizer）

不同于C++，Java没有真正意义上的析构函数，原因是所有著名的JVM实现都有垃圾收集器。无法保证对象一定会被垃圾收集器收集，因为这取决于程序中是否有指向它的引用以及其他一些随JVM实现而异的因素。

在对象即将被垃圾收集器收集时，垃圾收集器必须调用一个方法——`finalize()`。`java.lang.Object`类包含方法`finalize()`，任何类都可重写它：

```
@Override
protected void finalize() {
}
```

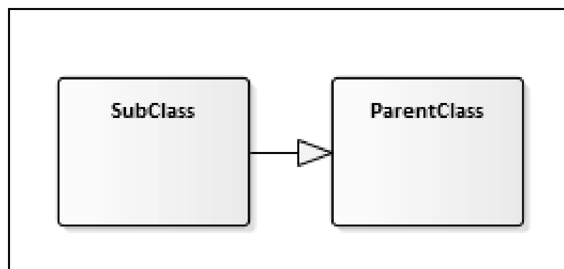
其中的语法`@Override`稍后就会介绍。

这个方法可用来释放类占用的资源，但仅当万不得已时才应这样做——建议程序员尽早关闭资源。鉴于无法保证方法`finalize()`一定会被调用，因此在代码的其他地方关闭或释放资源要好得多。

有些程序员在方法`finalize()`必须释放占用的资源时记录一条日志消息（或使用简单的`System.out.println()`调用），因为这通常昭示着程序存在bug——原本应该早已在其他地方释放了这些资源。

3. 扩展类

前面说过，JVM是一种只支持单继承的平台：



要继承类，可使用如下语法：

```
class SubClass extends ParentClass {
}
```



别忘了，指定了非访问限定符`final`的类是不能扩展的。

子类可访问它能够看到的父类的成员（变量和方法），而能否看到是由前面讨论类访问限定符时介绍的规则决定的。

(1) 重写方法

要访问父类的成员，可使用关键字`super`：

```
class TheParentClass {
  void aMethod() {
  }
}

class TheSubClass extends TheParentClass {
  @Override
  public void aMethod() {
    super.aMethod();
    // 其他代码……
  }
}
```

重写方法时，必须考虑其可见性。Java要求方法重写后的可见性不能降低；对于父类中受保护的方法，在子类中重写后可以是受保护的，也可以是公有的，但不能是私有或包私有的，因为这将降低可见性。前面的代码片段演示了这种概念：方法`aMethod()`在父类中是包私有的，而在子类中是公有的。

根据约定，应给重写的方法添加注解`@Override`，让人很容易知道它重写了一个既有的方法。将这个注解用于非重写方法时，编译器将报错。



注解旨在提供信号。就注解`@Override`而言，它旨在提醒阅读源代码的开发人员。也有由编译器或框架进行处理的注解。

(2) 调用父类的构造函数

要调用父类的构造函数，也可使用关键字`super`：

```
class A {
  public A(int i) { }
}

class B extends A {
  B(int i) {
    super(i);
  }
}
```

对于构造函数，Java有如下规则。

- 子类的每个构造函数都必须显式或隐式地调用父类的一个构造函数。
- 如果程序员没有给子类定义任何构造函数，父类必须包含一个不接受任何参数的构造函数。在这种情况下，Java将负责隐式地调用这个构造函数。
- 如果父类没有不接受任何参数的构造函数，子类的每个构造函数都必须使用有效的参数显式地调用父类的构造函数，且必须在子类构造函数的函数体中首先这样做。
- Java自动调用父类的不接受任何参数的构造函数（如果父类有这样的构造函数），但程序员也可手动这样做。这种调用必须是构造函数中的第一个非注释行。

下面通过一些示例来阐明上述规则。首先，来看父类和子类没有显式构造函数的情况：

```
class A { }
class B extends A { }
```

这两个类都没有提供任何构造函数，因此Java将自动为它们创建一个不接受任何参数的公有构造函数：A()和B()。生成的构造函数B()将自动调用构造函数A()，程序员也可手动显式地调用这个构造函数，但这种调用必须是构造函数中的第一条语句，否则Java编译器将拒绝对代码进行编译。在这种情况下，Java将不会隐式地调用父类的构造函数：

3

```
class A { }
class B extends A {

  public B() {
    super();
  }
}
```

在下面的示例中，父类包含默认、公有、不接受任何参数的构造函数，而子类包含一个接受参数的公有构造函数：

```
class A { }

class B extends A {
  public B(int i) {
  }
}
```

A没有提供构造函数，因此Java将自动创建公有构造函数A()。B指定了一个构造函数，因此它不会有隐式、公有、不接受任何参数的构造函数。虽然B类的构造函数没有显式地调用构造函数A()，但Java依然会在使用构造函数B(int i)时调用构造函数A()。

在下面的示例中，父类有一个接受参数的构造函数：

```
class A {
  public A(String s) {
  }
}

class B extends A {
  public B() {
    super("Hello");
  }
}
```

这里的情况比较有趣。父类只有一个带参数的构造函数，这意味着Java无法在子类中隐式地调用这个构造函数，因为它不想去猜测该将什么样的值传递给这个构造函数。现在，如果子类的构造函数没有显式地调用父类的构造函数，编译器将报错。另外，这种调用必须是子类构造函数的第一条语句，否则编译器也会报错。

4. 抽象类

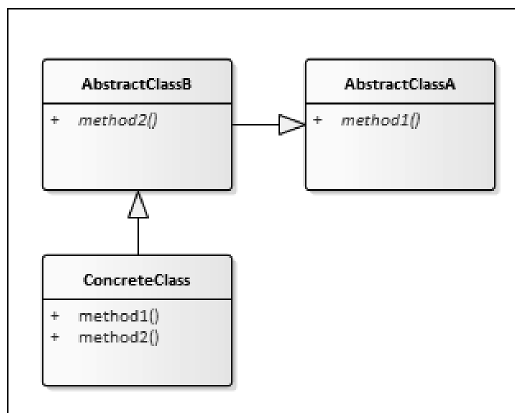
普通类被称为具体类。通过在类名前面加上非访问限定符abstract，可将类变成抽象类。

抽象类可被其他类扩展，但不能直接进行实例化。相比于具体类，抽象类的一个不同之处在于，它可以不给方法提供实现：

```
public abstract class AnAbstractClass {
  abstract public void thisIsAnAbstractMethod();
}
```

必须给类本身以及每个没有实现的方法都指定限定符abstract。抽象类可以包含具体方法，且并非必须包含抽象方法。在扩展抽象类的具体类中，必须通过重写来为所有抽象方法提供实现。

与具体类一样，抽象类也可继承另一个类，而继承的类可以是具体类，也可以是抽象类，如下所示：



如果将这个示例转换为Java代码，结果将类似于下面这样：

```
abstract class AbstractClassA {
  public abstract void method1();
}

abstract class AbstractClassB extends AbstractClassA {
  public abstract void method2();
}

class ConcreteClass extends AbstractClassB {
  @Override
  public void method1() { } // implementation code...

  @Override
  public void method2() { } // Implementation code...
}
```

由于抽象类AbstractClassB继承了抽象类AbstractClassA，因此ConcreteClass为这两个抽象类中的抽象方法提供实现。抽象类不能实例化，但引用变量可指向直接或间接扩展了抽象类的类：

```
AbstractClassA demo = new ConcreteClass();
```

demo只能用来访问AbstractClassA类的成员，而不能用来访问ConcreteClass的其他变量和方法。稍后你将看到，可将引用demo向下转换为ConcreteClass类型。

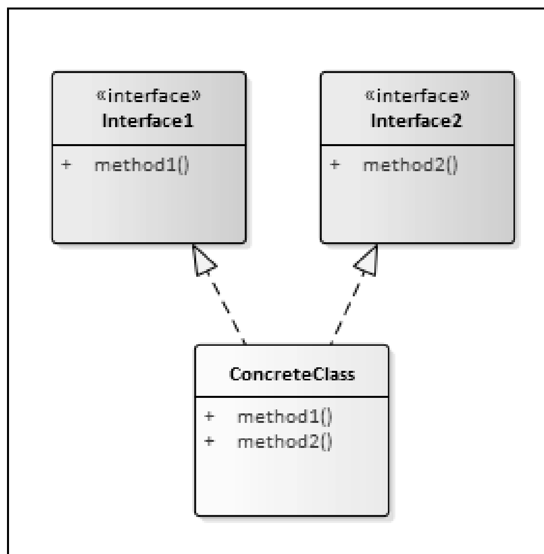
5. 接口

接口有点像抽象类。在Java 8之前，接口与抽象类的最大不同在于，接口不能为其任何方法提供实现。类不是扩展接口，而是实现它们。



稍后你将看到，在Java 8中，接口可给其方法提供默认实现。

在下面的示例中，一个类实现了两个接口：




```
interface Interface1 {
  void method1();
}

interface Interface2 {
  public void method2();
}
```

```
class ConcreteClass implements Interface1, Interface2 {
  @Override
  public void method1() { } // 实现代码……

  @Override
  public void method2() { } // 实现代码……
}
```

在源代码方面，接口遵守的规则与类相同。公有接口必须在与其名称完全相同的源代码文件中定义，因此一个源代码文件只能定义一个公有接口。另外，接口支持的访问限定符与类相同（公有和包私有）。

接口的成员默认为抽象和公有的。你可显式地添加关键字abstract和/或访问限定符public，但即便你省略限定符public，成员依然是公有的，而不是你可能认为的包私有的。接口成员只能是公有的，使用其他任何访问限定符都将导致编译器报错。

接口可包含方法和变量，但它们在接口中存在如下不同。

- 接口中的抽象方法总是公有的实例方法。
- 而接口中的变量总是final和静态的。你也可以显式地指定限定符final和static。

无论是具体类还是抽象类，都可根据需要实现任意数量的接口，但只有具体类必须通过重写来为所有方法提供实现。

在前面的示例中，ConcreteClass实现了接口Interface1和Interface2，因此必须重写这两个接口中的方法。虽然在接口Interface1中，没有给方法method1()指定限定符public，但它默认为公有的。引用类型变量可指向实现了特定接口的类：

```
Interface2 i = new ConcreteClass();
```

前面说过，从Java 8起，接口可为其方法提供默认实现。之所以添加这项新功能，是因为给既有类添加方法时，将导致接口与实现了它的既有类不兼容，因此这些类必须修改才能通过编译。现在，可以提供默认实现了，这样修改接口后，实现了它的既有类依然将与其兼容。

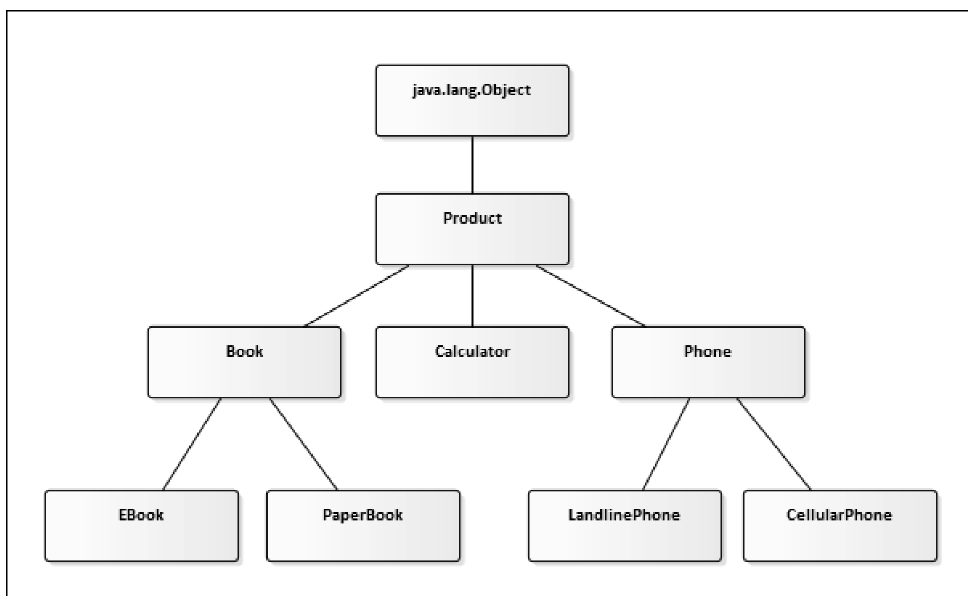
下面是一个这样的示例：

```
interface ExistingInterface {
  public void methodWithoutImplementation();
  default public void methodWithImplementation() {
    // 实现……
  }
}
```

另外，在Java 8中，可给接口添加静态方法。接口必须给静态方法提供实现，因为静态方法只能通过接口名来使用——通过引用类型变量无法访问接口的静态方法。

3.1.9 向上转换和向下转换

Java是一种静态类型的OOP语言。对象可转换为相关的类型。请看下面的类图：



由于在JVM中，`java.lang.Object`类是所有类的祖先类，因此包括这个类图中所有对象在内的每个对象都是`java.lang.Object`。同理，`Book`实例都是`Product`，`Calculator`和`Phone`实例亦如此。然而，`Product`实例不一定是`Book`实例：它可能是`Book`、`Calculator`、`Phone`或其任何一个子类的实例；如果不查看代码或文档，我们无法知道它是哪个子类的实例。

Java也面临这样的困境。我们说一个`PaperBook`实例是`Product`类的一个实例时，其实是将PaperBook向上转换成了`Product`实例。在代码中，转换类似于下面这样：

```
PaperBook paperBook = new PaperBook();
Product product = paperBook;
```

Java编译器知道，`PaperBook`实例总是可向上转换为`Product`实例，因此第二行代码能够通过编译。如果编译器不认同转换，它将拒绝对代码进行编译。因此，编译器将拒绝编译下面的代码：

```
Product product = new PaperBook();
PaperBook paperBook = product; // 这行代码不能通过编译
```

第一行没问题——`PaperBook`实例可自动向上转换为`Product`实例。但第二行不能通过编译，编译器显示的错误消息为incompatible types: Product cannot be converted to PaperBook。编译器只考虑变量`product`包含的是一个`Product`实例这一点，同时编译器不能保证`Product`实例都

能成功地向下转换为PaperBook实例，因此它必须让程序员知道，这种转换可能失败。要让上述代码能够通过编译，必须做如下修改：

```
Product product = new PaperBook();
PaperBook paperBook = (PaperBook)product;
```

现在，这些代码能够通过编译。在运行阶段，其中的Product实例将被向下转换为PaperBook实例。然而，如果我们犯了错，这样的转换将在运行阶段引发异常：

```
Product product = new PaperBook();
Phone phone = (Phone)product;
```

由于引用类型变量product指向的是一个PaperBook实例，因此无法将其向下转换为Phone实例。在运行阶段，JVM将引发异常ClassCastException：

```
Exception in thread "main" java.lang.ClassCastException: PaperBook
cannot be cast to Phone
```

如果你试图执行根本不可能的转换，编译器将拒绝编译代码，如下所示：

```
LandLinePhone landlinePhone = new LandLinePhone();
CellularPhone cellularPhone = (LandLinePhone)landlinePhone;
// 无法通过编译
```

Java编译器很聪明，知道LandLinePhone实例不可能是CellularPhone实例，因此拒绝编译这些代码。

3.2　编写 Java 代码

讨论Java所有相关的OOP功能后，就可以着手编写能够做些事情的类了。本节讨论一些可引导你完成这个过程的主题：

- 运算符；
- 普通的老式Java对象（POJO）；
- 数组；
- 泛型和集合；
- 循环；
- 异常；
- 线程；
- lambda。

3.2.1　运算符

下表总结了Java语言中一些最重要的运算符。请注意，除这里列出的运算符外，Java还支持

其他运算符。这里没有列出在其他所有流行的编程语言中都很常见的运算符，如+、-、>、>=、<和<=。

<table>
<tr><th>运 算 符</th><th>描 述</th></tr>
<tr><td><code>value++</code>
<code>value--</code></td><td>返回value，再将value的值加1或减1</td></tr>
<tr><td><code>++value</code>
<code>--value</code></td><td>将value的值加1或减1，再返回value的新值</td></tr>
<tr><td><code>!</code></td><td>逻辑NOT运算符</td></tr>
<tr><td><code>%</code></td><td>计算除法运算的余数</td></tr>
<tr><td><code>instanceof</code></td><td>返回一个布尔值，指出传入的对象是否是指定类或接口的实例</td></tr>
<tr><td><code>==</code>
<code>!=</code></td><td>相等和不等</td></tr>
<tr><td><code>&&</code>
<code>||</code></td><td>逻辑AND和OR运算符</td></tr>
<tr><td><code>=</code></td><td>赋值</td></tr>
<tr><td><code>+=</code>
<code>-=</code>
<code>*=</code>
<code>/=</code>
<code>%=</code></td><td>这些运算符计算新值并将结果直接赋给变量</td></tr>
</table>

下面是一些运算符使用示例：

```
class OperatorDemo {
  public static void main(String[] args) {
    int i = 0;
    System.out.println(i++);
    System.out.println(++i);
    System.out.println(i += 10);
  }
}
```

运行时，这些代码在控制台中打印如下输出：

```
0
2
12
```

由于`int`变量`i`是在方法中声明的，因此必须显式地初始化（在这里，其初始值为`0`）方法中的第二行代码打印`i`的当前值（`0`），再默默地将其增加到`1`。接下来的一行代码将变量`i`的值加`1`，再打印它（因此打印的为`2`）。最后将`i`的值加`10`，并打印结果（`12`）。

3.2.2 条件检查

条件检查方式有两种：

- ❑ if...else语句；
- ❑ switch...case语句。

1. **if...else**语句

在Java中，if和if...else语句没有什么让人意外的地方。每个条件都必须返回一个布尔结果，而else部分是可选的：

```
if (condition) {
} else if (condition) {
} else {
}
```

可像下面这样使用逻辑AND（&&）和OR（||）运算符来指定条件：

```
if (i > 25 || i == -1) {
}
```

用于基本类型变量时，==运算符与预期一致，但用于对象时，它检查两个对象引用是否相同，而不是比较它们的内容（属性）。因此，要检查String变量的内容，必须使用来自java.lang.Object类并经过重写的方法equals()，而不能使用运算符==：

```
String foo = "hello";
String bar = "world";
if (!foo.equals(bar))
  System.out.println("Not equal!");
```



大多数IDE都能够识别将运算符==用于String可能是错误的做法，并使用String的方法equals来重写代码。

2. **switch...case**语句

与很多其他的编程语言一样，Java也支持switch语句，下面是一个简单的示例：

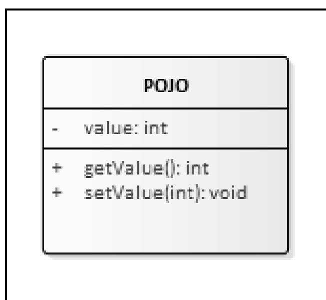
```
int value = 3;
String s = "";
switch (value) {
  case 1:
    s = "One";
    break;
  case 2:
  case 3:
    s = "Two or three";
    break;
  default:
    s = "Something else";
}
System.out.println(s);
```

对于switch语句的用法，有几点需要说明。

- 指定的表达式的结果可以是整数，也可以是字符串。
- 在case语句中指定的值必须是在编译阶段能够确定的，因此非final变量不能用来指定这种值。
- 必须在case语句中添加一条break语句来跳到switch语句末尾，否则将接着执行下一条case语句。

3.2.3 POJO

最初，没有支持Java语言的框架，开发人员大多自己编写类。在Java的发展过程中，引入了很多框架，而其中很多都要求类实现特定的接口或扩展框架提供的类。这导致类与特定的框架紧密耦合，而编写的代码无法在使用其他框架的项目中重用。对于这样的情形，并非每个人都感到满意，因此不久后一种新趋势开始流行：重回普通的**老式Java对象**（Plain Old Java Objects，POJO）。当前，很多流行的框架都支持POJO：



一个类如果满足如下条件，就被视为真正的POJO：

- 没有扩展类，也没有实现接口；
- 是可修改的；
- 包含一个不接受任何参数的公有构造函数；
- 使用公有方法来存储和获取私有变量的值。



即便需要扩展类或实现接口，遵守POJO的其他设计规则也是不错的选择。POJO不是必须遵守的规则，而是一种约定。

下面就是一个POJO类：

```
class POJO {
   private int value = 0;

   public POJO() {
```

```
    }

    public int getValue() {
      return value;
    }

    public void setValue(int value) {
      this.value = value;
    }
}
```

在POJO类的实例中，可使用被称为**属性**的方法来存储和获取值。对于每个属性，都遵守如下约定：

- 其值存储在私有变量中；
- 有一个用于返回其值的公有获取方法；
- 有一个用于存储其值的公有设置方法。

获取方法（getter）的名称以`get`打头，后面通常是变量名。如果变量是布尔类型，获取方法通常以`is`（而不是`get`）打头。设置方法（setter）的名称通常以`set`打头，而不管变量是哪种类型。

通常，再添加一个公有的重载构造函数，它将POJO类的所有属性（这里是`value`）作为参数：

```
public POJO(int value) {
  this.value = value;
}
```



在大多数IDE中，都只需单击一个按钮就能生成POJO或给既有POJO添加属性。

3.2.4　数组

Java提供了内置的数组支持。要声明数组，可在类型或变量名后添加`[]`，并使用关键字`new`来创建数组并设置其长度：

```
int[] intArray1 = new int[2];
int intArray2[] = new int[2];
```

在Java中，必须在创建数组时显式地设置其长度。对于基本类型数组，其元素的默认初始值为0（数值类型）或false（布尔类型），而对于引用类型数组，其元素的默认初始值为null。

与很多其他流行的编程语言一样，索引是从零开始的。在前面的示例中，数组`intArray1`和`intArray2`的索引可以是`0`或`1`。如果使用的索引超出了范围，将引发运行阶段异常：

```
intArray1[0] = 10;
intArray1[1] = 20;
```

要获取数组的长度，可使用数组提供的只读变量length：

```
System.out.println(intArray1.length);
```

上述代码将在控制台中打印2。数组缺少一些方便的功能，例如，它们没有重写方法toString()，因此使用方法System.out.println打印数组变量intArray1，得到的将是Object提供的默认输出，如[I@659e0bfd，这没有提供有关数组内容的任何信息。

java.util包中有一个实用的Arrays类，这个类包含很多便利的静态方法。如果你要将数组转换为集合类的实例、在数组中查找元素、对数组进行排序等，建议你参阅API文档。下面是一个使用java.util.Arrays类的示例：


```
System.out.println(java.util.Arrays.toString(intArray1));
```

这将打印[10, 20]。

可在声明数组的同时对其进行初始化：

```
int[] intArray = { 10, 20, 30 };
```

在这个示例中，Java编译器将自动声明一个包含3个元素的int数组，并将每个元素都初始化为指定的值。

3.2.5 泛型和集合

前一章讨论了集合，因为它们对JVM来说非常重要。刚引入Java时，集合类只能存储java.lang.Object对象。鉴于JVM中的每个对象都可向上转换为java.lang.Object实例，因此集合一直能够存储任何类型的对象。这种灵活性也有缺点：要访问类成员，必须先向下转换对象。如果不小心添加了一个不同类型的对象，这可能导致运行阶段错误。下面的示例程序能够通过编译，但运行时将导致错误：

```
import java.util.ArrayList;

class ClassCastExceptionExample {
  public static void main(String[] args) {
    ArrayList list = new ArrayList();

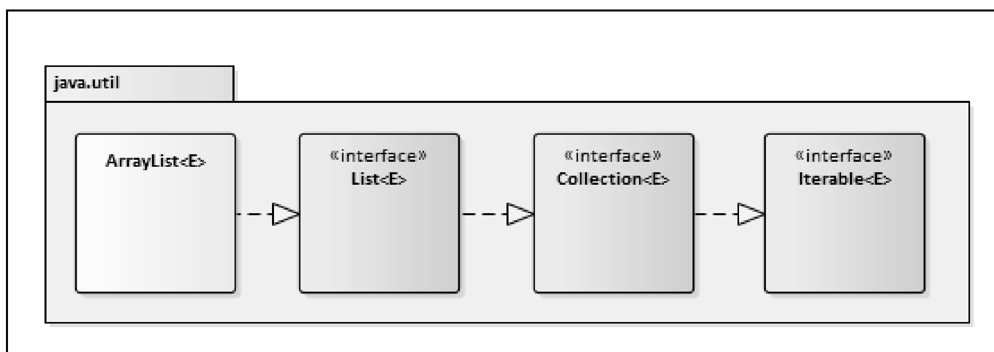
    list.add(new Integer(123));
    list.add("This is not an integer");

    Integer i = (Integer)list.get(0);
    i = (Integer)list.get(1); // 运行阶段异常!!!
  }
}
```

不能将java.lang.String实例转换为java.lang.Integer实例，因此将第二个元素（索

引为1）转换为整数时，引发了ClassCastException异常。

为确保某些类只能用于存储固定的类型（由开发人员指定），在Java语言中添加了泛型。例如，可创建一个只能存储java.lang.Integer实例的ArrayList对象，这样如果代码试图在这个ArrayList对象中添加其他对象，编译器将拒绝编译。泛型是一个复杂的主题，这里只讨论其基本用法。鉴于后面的示例将使用ArrayList，因此先来看下面的类图，其中显示了ArrayList实现的一些接口。



<E>表示相应的接口（和实现它的类）支持泛型。可将E视为使用List时将指定的类型的别名。根据约定，使用单个字母（这里是E）来表示元素。要声明一个只能存储java.lang.Integer实例的java.util.ArrayList实例，可使用一个指向List接口的引用：

```
import java.util.ArrayList;
import java.util.List;

class GenericsExample {
  public static void main(String[] args) {
    List<Integer> listWithIntegers = new ArrayList<>();
    listWithIntegers.add(new Integer(1));
  }
}
```

为指定所需的类型（这里是Integer类），在接口类型List后面添加了<Integer>。创建ArrayList的实例时，无需再指定Integer类，而只需添加<>即可。现在，如果你试图添加不能向上转换为Integer的类的实例，编译器将拒绝对代码进行编译。如果不使用泛型，这种错误只能到运行阶段才能检测到，因此程序员大都认为这是一种进步。

我们让引用类型变量listWithIntegers指向接口java.util.List，而不是ArrayList类，虽然并非必须这样做。java.util.List是一个泛型接口，ArrayList以及Collection API中的其他数据结构都实现了它。这样做是一种不错的约定，因为通过这样做，可将ArrayList替换为实现了接口java.util.List的其他数据结构，而无需对其他代码做任何修改。



在JVM领域，隐藏实现细节是一种非常好的设计选择，而接口和抽象类让这成为可能。

下面来看一个使用泛型HashMap的示例。HashMap是一个位于java.util包中的类，实现了更通用的接口java.util.Map。同样，这里也将把引用变量的类型声明为Map接口，以隐藏这样的设计信息，即我们使用的是HashMap类。我们先来看看接口Map：

```
public interface Map<K,V>
```

从中可知，Map支持泛型，且要求指定两种类型——K和V（它们分别代表键和值）。下面来创建一个将String键映射到Integer值的HashMap实例：

```
import java.util.HashMap;
import java.util.Map;

class GenericsExample {
  public static void main(String[] args) {
    Map<String, Integer> map = new HashMap<>();
    map.put("one", new Integer(1));
    map.put("ten", new Integer(10));
    System.out.println(map.get("one"));
  }
}
```

这将向控制台打印Integer值1。



泛型只适用于对象，指定基本类型值将导致编译错误。处理泛型时，编译器不会将基本类型值自动装箱，也不会将对象自动拆箱。

3.2.6　循环

数组和集合很好，但仅当你能够遍历它们时才如此。Java提供了如下内置的循环结构：

- for循环；
- while循环。

for循环

Java有两种for循环：

- 常规for循环（使用计数器）；
- 改进的for循环（用于对象）。

(1) 常规for循环

常规for循环的语法如下：

```
for (int i=0; i < 10; i++) {
  System.out.println(i);
}
```

这将打印0–9，每个数字各占一行。

与其他语言中一样，for循环由三部分组成。

- 第一部分初始化计数器。
- 第二部分包含每次迭代开始前都将检查的表达式。如果这个表达式返回false，将停止迭代。
- 最后一部分是每次迭代后都将调用的语句。

每部分都是可选的，但分隔各部分的分号必不可少。一种不同寻常的结构是无限循环，可通过将每部分都设置为空来创建：

```
for (;;) {
}
```

可使用关键字break来提前结束for循环。要结束当前迭代并进入下一次迭代，可使用关键字continue：

```
for (int i=0; i < 4; i++) {
  if (i == 1)
    continue;
  if (i == 3)
    break;
  System.out.println(i);
}
```

上述for循环在i为1时跳到下一次迭代，并在i为3时结束，因此它只向控制台打印0和2。

(2) 改进的for循环

改进的for循环只能用于数组和实现了泛型接口java.lang.Iterable<T>的对象。大多数Collection API类都实现了这个接口。下面是一个将这种for循环用于数组的示例：

```
String[] stringArray = { "One", "Two", "Three" };
for (String s: stringArray) {
  System.out.println(s);
}
```

这将打印One、Two和Three，每个字符串各占一行。推荐你尽可能使用这种for循环，因为这样可提高代码的可读性。

(3) while循环

在Java中，while循环类似于下面这样：

```
int i = 10;
while (i < 10) {
  System.out.println(i);
  i++;
}
```

其中的表达式的结果必须为布尔值。这个示例什么都不会打印，因为int变量i的值为10，导致表达式的结果为false，所以根本就不会进入循环。

与for循环一样，while循环也可使用关键字break来提前结束，还可使用关键字continue来跳到下一次迭代。

(4) do...while循环

do...while循环很像while循环，唯一的差别是它在迭代结束后再评估表达式。在任何情况下，都将进入这种循环：

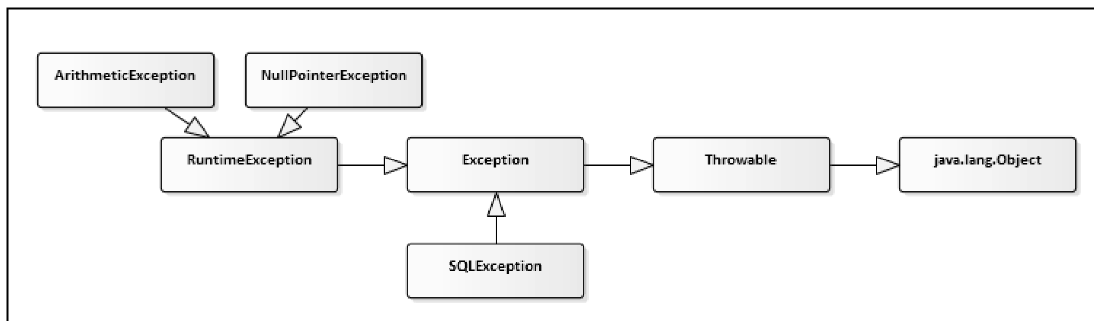
```
int i = 10;
do {
  System.out.println(i);
  i++;
} while (i < 10);
```

这将打印10。在第一次迭代结束后对表达式进行评估；由于11大于10，因此表达式返回false，循环就此结束。

与for循环和while循环一样，do...while循环也可使用关键字break来提前结束，还可使用关键字continue来跳到下一次迭代。

3.2.7 异常

异常在前一章讨论过。要处理异常，必须将代码放在try和catch块之间。可指定多种要处理的异常。下面的类图列出了几种异常：



下面的示例代码故意引发了除零错误：

```
try {
  System.out.println(0/0);
  System.out.println("exit");
} catch (NullPointerException e) {
  System.out.println("NULL POINTER EXCEPTION!");
} catch (ArithmeticException e) {
  System.out.println(e.getMessage());
  e.printStackTrace();
} catch (Exception e) {
  System.out.println("DIFFERENT ERROR");
}
```

这将打印如下内容：

```
/ by zero
java.lang.ArithmeticException: / by zero
at JavaApplication8.main(JavaApplication8.java:4)
```

注意，没有打印文本`exit`，因为相应的语句还未执行就引发了异常。表达式`0/0`引发异常`ArithmeticException`后，JVM分析错误处理程序中的所有catch块。由于引发的异常`ArithmeticException`不是`NullPointerException`的子类，因此忽略第一个catch块。第二个catch块与这个异常完全匹配，因此执行这个catch块，打印这个异常的消息和栈跟踪。

如果引发的异常不是`NullPointerException`或`ArithmeticException`，也不是它们的子类，JVM将指定第三个catch块，因为异常通常是`java.lang.Exception`的子类。

如果这些catch块都与引发的异常不匹配，JVM将查看方法的调用者。如果其中有`try...catch`块，就对其进行分析。如果找到匹配的catch块，就执行它。如果调用者没有`try...catch`块，就查看调用者的调用者，直到找遍调用栈。如果没有`try...catch`块能够处理错误，程序将崩溃。

必须按正确的顺序放置catch块：将捕获最具体的异常类的catch块放在最前面，然后是捕获`Exception`子类的catch块，最后是捕获`Exception`本身的catch块。下面的示例不能通过编译：

```
try {
  Integer i = null;
  System.out.println(i.toString());
} catch (Exception e) {
  // ...
} catch (NullPointerException e) {
  // 不能通过编译！
}
```

因为异常`NullPointerException`是`Exception`类的子类，Java编译器知道不可能到达捕获`NullPointerException`的catch块，所以拒绝编译这些代码。

可添加可选的`finally`块。在任何情况下，`finally`块中的语句都会执行，即便引发了异常：

```
try {
  throw new Exception("oops");
} catch (Exception e) {
  System.out.println("Exception!");
} finally {
  System.out.println("FINALLY!");
}
```

这将打印如下内容：

```
Exception!
FINALLY!
```


运行阶段异常

在方法可引发的异常方面，Java的要求不同寻常：方法可引发任何属于java.lang.RuntimeException的子类的异常。从本章前面的类图可知，NullPointerException和ArithmeticException都是RuntimeException的子类。

要引发不是java.lang.RuntimeException的子类的异常，方法必须显式地列出它们。来看一下接口java.sql.ResultSet的一个方法的声明：

```
public int getInt(String column) throws SQLException
```

这个方法将一个表示列名的String作为参数，并返回从数据库中检索到的int值。这个方法的签名告诉Java编译器，它可能引发SQLException异常，这个异常是java.lang.Exception的子类，但不是java.lang.RuntimeException的子类。现在，Java编译器知道，这个方法可能引发SQLException异常。

重写方法时，无论方法来自具体类、抽象类还是接口，均可：

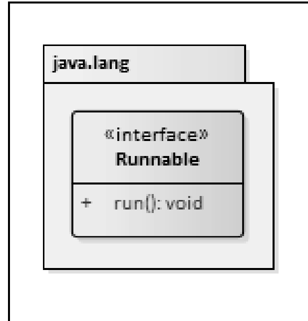
- 选择不引发任何异常，为此可根本不添加关键字throws；
- 使用关键字throws接受原始方法的全部或部分异常；
- 将throws列表中的部分或全部异常替换为其子类。

除非原始方法可能引发的异常为RuntimeException，否则重写后的方法不能引发其他异常。

3.2.8 线程

这里只介绍最简单的并发编程：启动多个线程。

在支持线程方面，接口java.lang.Runnable扮演着至关重要的角色。这个接口很简单，只包含一个方法：



如果一个类实现了这个接口并提供了方法run()的实现，就可在独立的线程中启动。来看一个简单的示例。

首先，有一个实现了接口Runnable的类，能够在不同的线程中运行：

```
class SleepyClass implements Runnable {
  private int number;

  public SleepyClass(int number) { this.number = number; }

  @Override
  public void run() {
    System.out.println("Thread " + number + " started!");
    try {
      Thread.sleep(3000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Thread " + number + " ended!");
  }
}
```

接下来，有另一个类，它在两个线程中运行前述代码：

```
public class ThreadsDemo {
  public static void main(String[] args) {
    Thread thread1 = new Thread(new SleepyClass(1));
    Thread thread2 = new Thread(new SleepyClass(2));
    thread1.start();
    thread2.start();
  }
}
```

这个程序运行时，输出可能是这样的：

```
Thread 1 started!
Thread 2 started!
Thread 2 ended!
Thread 1 ended!
```

对线程的启动顺序，JVM不做任何保证，即不保证线程1先开始、先结束。如果CPU内核足够多，可能每个线程都会在不同的CPU内核中运行。

`SleepyClass`类展示了Java开发人员经常会遇到的与异常处理相关的问题。`Thread`类的方法`sleep`可能引发异常`InterruptedException`，这种异常不是`RuntimeException`的子类。我们不能在方法`run()`的声明中添加`throws InterruptedException`，因为接口`Runnable`的方法`run()`不会引发这种异常或其子类。因此，我们选择在`run()`中处理这种异常。

并发编程很难。程序员必须确保多个线程不会同时修改同一个变量，否则可能引发竞态条件，因而导致数据受损和微妙的bug。之所以会出现这些问题，是因为对变量的操作大都不是原子性的，为完成一个操作必须执行多条CPU指令。在一个操作还在进行时，另一个线程将开始对同一个变量执行操作。

Java提供了一个用于方法的非访问限定符——`synchronized`，可确保方法不会被多个线程同时调用：其他线程等待当前线程调用完方法，再依次调用这个方法：

```
public synchronized void synchronizedMethod() {
}
```

线程调用完方法后，JVM将确保相应的锁得以释放，即便方法引发了异常。



不推荐大量地使用限定符`synchronized`，因为锁定线程是一种开销很大的操作，可能严重降低程序的性能。

如果你对并发编程感兴趣，请详细研究`java.util.concurrent`包中的类。

3.2.9 lambda

在Java 8新增的功能中，lambda可能是最受欢迎的，它们让你能够将函数像变量一样传递给方法。在本书介绍的很多语言中，都可使用lambda，因为很多其他的语言都提供了内置的lambda支持。

要向函数传递lambda，得先创建一个函数式接口（functional interface）。函数式接口是只包含一个抽象方法的普通接口。Java自带了大量支持lambda的接口，其中之一是接口`java.lang.Runnable`。由于这个接口只包含抽象方法`run()`，因此非常适合用来支持lambda。如果你只想运行一个线程，可直接向`Thread`实例传递一个匿名的lambda函数：

```
public class LambdaDemo {
  public static void main(String[] args) {
    Thread thread1 = new Thread( () -> {
      try {
        Thread.sleep(3000);
      } catch (InterruptedException e) {
```

```
            }
        });
        thread1.start();
    }
}
```

乍一看，其中的语法可能令人迷惑。考虑到接口`java.lang.Runnable`的方法`run()`不接受任何参数，我们首先指定一对空括号（`()`），再指定一个箭头（`->`），它告诉编译器接下来是一个lambda。

在这个代码块中，像通常那样编写将使用线程来运行的函数的代码。

在下一章，我们将使用更复杂的接受参数的lambda。

3.3 编程风格指南

到目前为止，Oracle没有提供与时俱进的官方Java语言编程风格指南，这方面的最新指南是Sun 公司于1999年发布的，本书编写期间还保留在Java网站上。下面是其中一些在当前依然适用的要点。

- 建议每个项目文件都以相同的注释打头，并在其中至少包含类名和版权信息。
- 公有类或公有接口必须放在文件的开头。如果文件还包含和具有其他访问限定符的其他类或接口，应将其放在公有类或公有接口后面。
- 按如下顺序定义类或接口。

  (1) 类或接口的Javadoc注释（这个主题将在下一章讨论）。
  (2) 关键字`class`或`interface`（包括访问限定符和其他限定符）。在接口名或类名后面，表示类代码块起始位置的`{`字符应与接口名或类名位于同一行，但表示类代码块结束位置的`}`字符独占一行。
  (3) 包含文档中没有的实现信息的文档。
  (4) 按如下顺序排列的静态变量：

    - 公有的静态变量；
    - 受保护的静态变量；
    - 包私有（没有访问限定符）的静态变量；
    - 私有的静态变量。

  (5) 实例变量（排列顺序与静态变量相同）。
  (6) 构造函数。
  (7) 方法（按功能而不是访问限定符分组）。

- 不要在同一行声明多个变量，而让每个变量声明都独占一行。
- 尽可能在声明变量的同时初始化它。
- 变量声明应放在用{ }括起的代码块的开头，而不要在代码块中间声明变量。

3.4 小测验

我们通过一个小测验来测试一下你的Java知识，答案见附录A。如果你得分很高，祝贺你！如果你做得不怎么样，也不用失望，只需再阅读相关的部分，然后看看你是否能做得更好。

(1) 下面的代码能够通过编译吗？如果不能，是什么地方有问题？

```
import java.util.ArrayList;
package com.example.quiz1;
public class Question1 {
}
```

a) 包名不对，因为包名不能包含数字。

b) ArrayList类不在java.util包中。

c) package语句必须放在import语句前面。

d) 以上说法都不对，这个文件能够通过编译。

(2) 下面的代码能够通过编译吗？如果不能，是什么地方有问题？

```
class A { }

class B { }

class C extends A, B {
}
```

a) 类名不能只包含一个字母。

b) 在Java中，一个类不能继承（扩展）多个类。

c) 在一个Java源代码文件中，不能定义多个类。

d) 以上说法都不对，这个文件能够通过编译。

(3) 下述代码的哪部分很可能有问题（假设这些代码放在一个有效的方法中）？

```
String s1 = "String A";
String s2 = "String B";

if (s1 != s2) {
  // 其他代码……
}
```

a) 程序员很可能想检查字符串的内容，因此应使用方法equals。

b) Java不支持运算符!=。

c) 在没有其他代码的情况下，无法判断这个代码片段是否存在逻辑错误。

d) 没什么问题，一切正常。

(4) 重写类或接口中的方法时，重写后的方法可随便决定要引发哪些异常吗？

a) 是。

b) 否。

(5) 下面的方法被调用时（假设已经将这个方法放到了正确的类中），将向控制台打印什么样的输出？

```
void testMethod() {
  try {
    System.out.println("A");
    throw new RuntimeException("Error!");
  } catch(Exception e) {
    System.out.println("C");
    return;
  } finally {
    System.out.println("D");
  }
}
```

a) A和C。

b) A和D。

c) A、D和C。

d) A、C和D。

3.5　小结

本章研究了大量的Java代码。我们首先介绍了Java的所有OOP功能，包括定义类、将类放到包中，以及通过定义新方法和变量给类添加成员。我们了解到，Java的面向对象功能并不止这些，因为抽象类和接口提供了很多确保代码结构良好的途径。我们讨论了最重要的访问限定符和非访问限定符，如何向上和向下转换类实例，以及POJO约定。最后，讨论了Java的各种重要功能，包括最重要的运算符、`if...else`语句、`for`循环、`while`和`do...while`循环。我们还介绍了数组、集合、泛型和异常。我们还尝试使用了一些较高级的功能，如多线程和lambda。

有了这些知识后，就可开发第一个Java项目了。让我们一起来编写一些代码！

第 4 章

Java编程

4

我们已经掌握大量的理论了，现在来编写一个真正的Java程序。我们将编写一个简单的独立Web服务，它计算输入字符串中每个字符的频度，并以JSON字典的方式返回结果。我们将使用构建工具Gradle从网络自动下载依赖项，再构建并运行项目。在编写后端类时，我们将使用测试驱动的方法，在编写代码的同时编写单元测试。在从编码到运行最终Web服务的每个阶段，我们都将使用Eclipse IDE。最后，我们将讨论各种尽可能提高效率的快捷方式。本章将介绍如下主题：

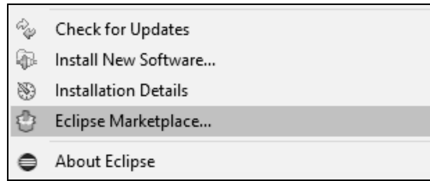
- 配置Eclipse IDE；
- 在Eclipse IDE中新建一个基于Gradle的项目；
- 修改Gradle构建脚本；
- 构建项目；
- 编写后端类；
- 创建单元测试对后端类进行测试；
- 编写Web服务；
- 运行Web服务。

4.1 配置 Eclipse IDE

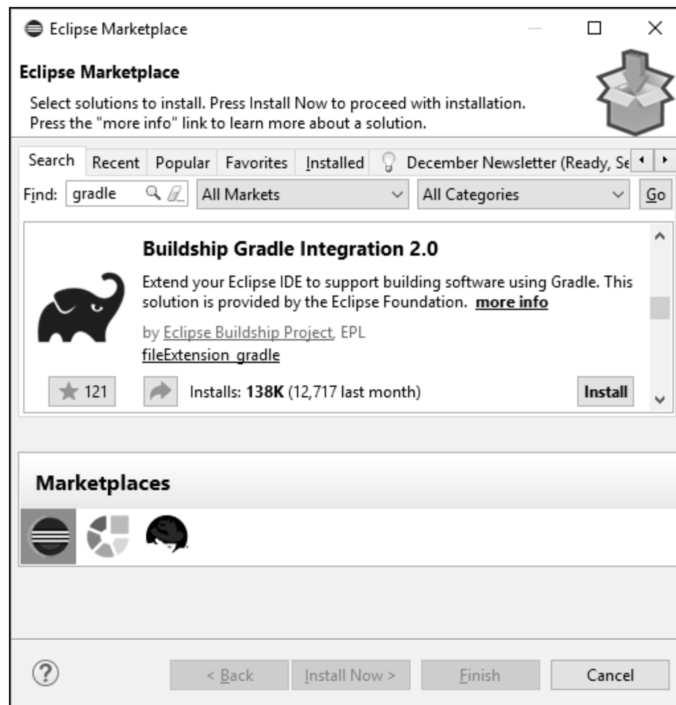
如果你使用Eclipse IDE中默认的项目类型Java Project，Eclipse将生成一个基于构建工具Apache Ant的XML构建脚本，并在你选择Compile或Build选项时执行其中的任务。对小型项目来说，这很好，但对于较大的项目，你通常需要更大的控制权。在这种情况下，我们希望Eclipse自动下载额外的依赖项。对于本章的项目，我们选择使用非常流行的构建工具Gradle。

由于Eclipse IDE没有提供内置的Gradle支持，我们需要安装一个在Eclipse IDE中添加这种功能的插件。可实现这个目标的插件有很多，我们将使用Gradle团队开发的插件。为安装这个插件，请按如下说明做。

- 选择菜单Help>Eclipse Marketplace...；



- 在文本框Find中输入`gradle`并按回车键。在找到的插件列表中向下滚动，直到看到列表项Buildship Gradle Integration 2.0。该列表项应带Gradle徽标（大象），且其中指出了由Eclipse Buildship Project开发。如果还没有安装它，请单击Install按钮：



- 如果你同意许可条款，接受条款并单击Finish按钮。
- 过段时间后，Eclipse IDE将显示一个对话框，指出必须重启该IDE。单击Yes确认要自动重启。

这样这个插件就安装好了，可以使用了。

4.2　使用 Java 创建 Web 服务

我们将使用Java以测试驱动的方法创建一个简单的Web服务。为此，我们将采取如下步骤：

- 在Eclipse IDE中创建项目；
- 修改Gradle构建文件；
- 编写后端类；
- 编写Web服务。

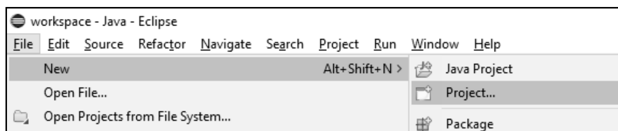
4.2.1 在 Eclipse IDE 中新建 Gradle 项目

我们将创建一个Gradle项目并查看生成的项目。

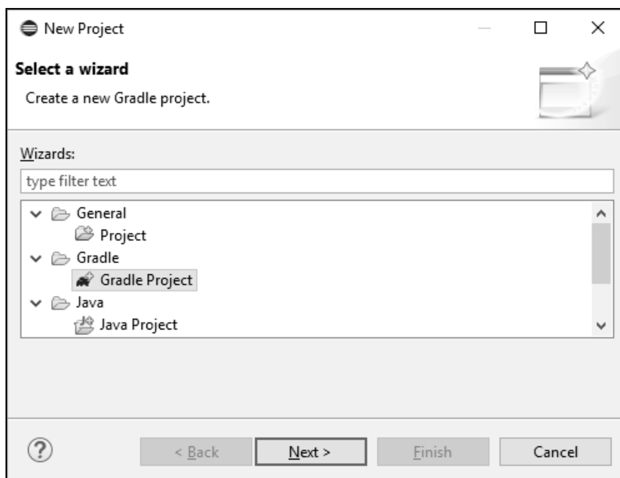
如果还没有启动Eclipse IDE，现在就启动它。如果必要，确认工作空间目录——Eclipse将在这个目录中创建和查找项目。

接下来，将出现Eclipse IDE Welcome选项卡。鉴于这个选项卡中没有新建Gradle项目的快捷方式，我们先不管它。为创建Gradle项目，请执行如下操作。

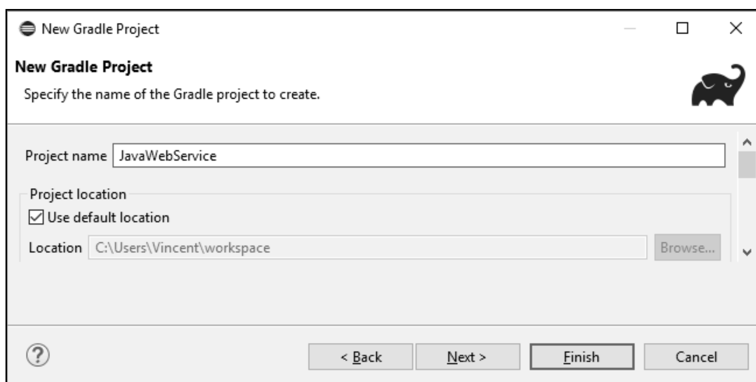
- 选择菜单File>New>Project...（千万不要选择Java Project，否则将不会使用Gradle来构建项目）：



- 在向导窗口New Project中，展开选项Gradle并选择Gradle Project，再单击Next按钮。然后将出现一个欢迎窗口，请仔细阅读其中的文本。如果愿意，可取消选择复选框Show the welcome page the next time the wizard appears。单击Next按钮：



- 在New Gradle Project窗口中，指定项目名——JavaWebservice，确保同意将项目存储到默认位置，再单击Finish按钮。第一次使用时，插件可能需要一段时间来下载并安装Gradle。最终，这个窗口将自动关闭。



请注意，在这个屏幕截图中，调整了窗口的大小，你看到的窗口将包含更多选项。

探索生成的项目

Eclipse IDE窗口左边将出现一个包含Package Explorer选项卡的窗口，请展开该选项卡中的项目JavaWebservice。我们来简单地看看这个生成的项目。

Gradle构建插件生成了如下项目条目。

- src/main是一个快捷方式，指向主程序源代码所在的目录。
- src/test是一个快捷方式，指向单元测试脚本所在的目录。
- JRE System Library显示了运行程序所需的Java平台文件。
- Project and External Dependencies显示了程序所需的附加库。当前，Gradle默认加载的单元测试框架JUnit 4需要多个库。
- 目录gradle包含Gradle wrapper所需的文件。这个脚本确保你能够在没有安装Gradle的系统中运行项目，并确保在构建项目期间下载并使用正确的Gradle版本。
- src显示了源代码目录的完整内容。当前，它包含子目录main和test，这在本书前面讨论过。
- 最后，在项目的根目录中，有一些与Gradle相关的文件，其中最重要的是build.gradle，Gradle将使用这个构建文件来编译和构建项目以及运行单元测试。

4.2.2　修改 Gradle 构建文件

对于这个项目，我们将使用框架SparkJava。第1章简要讨论过SparkJava，请不要将其与Apache的大数据平台混为一谈。SparkJava是一个让你能够轻松创建快速、独立Web应用程序的框架。你

可手动从官方网站下载这个库，并将必要的文件放到正确的目录中，但让构建工具去处理这些事情要容易得多。



很多著名的库都依赖于众多其他的依赖项，而这些依赖项也有自己的依赖项。对现代JVM软件开发来说，能够下载库的构建工具是必不可少的。

双击Package Explorer中的文件build.gradle将其打开。最新的Gradle版本使用基于Groovy的**领域特定语言**（domain-specific language，DSL）；以后的版本将支持基于Kotlin的DSL。你将在第11章看到，Groovy对语法的要求没有Java严格，因此在Gradle构建文件中，分号和小括号通常并不是必不可少的。找到`dependencies`块：

```
dependencies {
  ....
}
```

如果其中包含以`compile`打头的行，请将它们删除，但务必保留以`testCompile`打头的语句。在`dependencies`块开头添加如下条目：

```
compile 'org.slf4j:slf4j-simple:1.7.21'
compile 'com.sparkjava:spark-core:2.5.4'
compile 'com.fasterxml.jackson.core:jackson-databind:2.8.5'
```

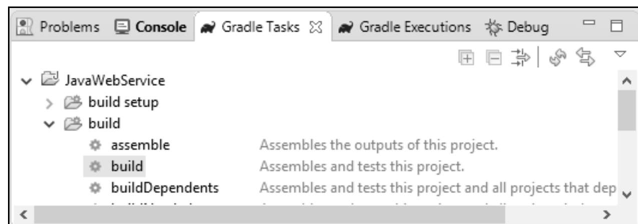
将修改后的文件存盘。这个文件告诉Gradle，为编译这个项目，需要Simple Logging Facade for Java（SLF4J）、SparkJava和Jackson's JSON handler框架。在依赖项中包含版本号通常是个不错的主意，因为更新的版本可能导致既有代码无法正确运行。构建项目时，Gradle将搜索流行的仓库网站，下载指定版本的依赖项及其依赖项，并将它们放到正确的目录中。还将设置classpath，让你无需手动修改它就能运行项目。



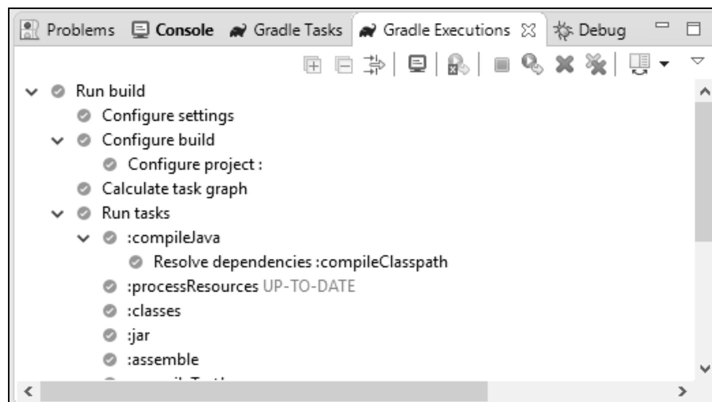
别忘了，较新版本的库和框架能修复重要的安全bug，这对生产环境来说尤其重要。在任何情况下，紧跟你依赖的框架的发展步伐都是明智的。

4.2.3　构建项目

在Eclipse IDE右下方的窗口中，找到并单击标签Gradle Tasks。展开条目build，再双击其中的build任务，如下图所示。



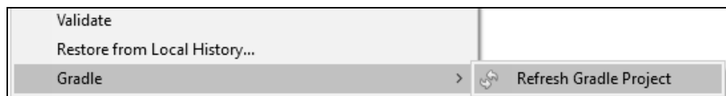
这将切换到Gradle Executions选项卡，其中显示了插件Gradle运行的任务的进度和状态。如果一切顺利，每项任务旁边都将出现绿点，如下图所示。



如果不太顺利，且只有任务:compileJava旁边为红点，请切换到选项卡Console，其中包含Gradle的输出。请向下滚动，看看能否找到这样的错误消息，即未能找到tools.jar。如果有这样的错误消息，就意味着你需要将Java安装位置告诉插件Gradle。为此，可按如下步骤做。

(1) 切换到选项卡Gradle Tasks。
(2) 右击任务build并选择Open Gradle Run Configuration...。
(3) 这将打开Edit configuration对话框。切换到选项卡Java Home并单击按钮Browse。切换到JDK安装目录，并单击OK按钮关闭对话框；再单击OK按钮关闭对话框Edit configuration。
(4) 双击任务build以再次执行它。现在选项卡Gradle Executions中将只有绿点。

Eclipse插件Buildship Gradle Integration的最新版存在的一个问题是，添加、更新或删除依赖后，你可能必须手动刷新项目。请在Package Explorer中展开条目Project and External Dependencies，如果其中没有列出JAR文件spark-core，请右击项目JavaWebservice并选择Gradle>Refresh Gradle Project。这样做后，将列出其他很多作为依赖的JAR文件。



4.2.4　编写后端类

这里将创建一个可重用的通用后端类，它对使用的Web服务技术一无所知。用于处理HTTP请求的代码将使用这个后端类来生成JSON响应。我们将使用**测试驱动开发**（test-driven development，TDD）方法，并涵盖如下主题：

- ❑ 后端类的业务规则；
- ❑ 创建方法的哑实现（dummy implementation）；
- ❑ 创建测试用例类并编写其第一个单元测试；
- ❑ 实现输入有效性检查；
- ❑ 编写第二个单元测试；
- ❑ 实现业务逻辑；
- ❑ 创建Web服务。

1. 后端类的业务规则

我们来创建一个简单的Web服务，它返回发送给它的字符串中各个字符出现的次数。

计算字符串中各个字符出现的次数的方法必须满足如下需求。

- ❑ 它所在的类必须放在`chapter03.backend`包中，同时必须名为`CharacterCounter`且是公有的。
- ❑ 这个方法必须名为`countCharacters`，它将一个`String`作为输入值且是公有的。
- ❑ 这个方法必须返回一个实现了泛型接口`Map<Character, Integer>`的类的实例。这是一个将`Character`实例映射到`Integer`实例的映射。
- ❑ 返回的`Map`对象必须将输入字符串包含的每个UTF-16字符映射到一个整数，而这个整数指出了这个字符在字符串中出现的次数。
- ❑ 传入的字符串不能是空的，否则将引发`IllegalArgumentException`异常。

下面是一个输入/输出示例：

```
"A!Ba?!?!" --> {'A': 1, 'B':1, 'a':1, '?': 2, '!': 3}
```

2. 创建方法的哑实现

要编写单元测试，必须先编写符合签名要求（输入和输出）的方法，但我们只是让这个方法返回null。这样做后，就可编写单元测试，并在确定测试失败后提供方法的正确实现，以确保测试成功。

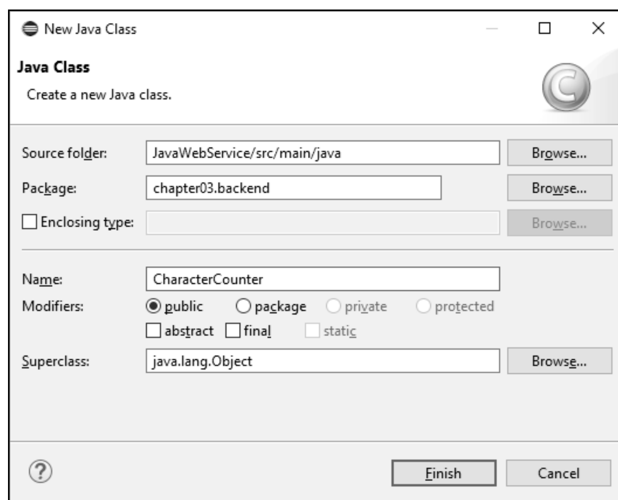
如果你详细研究业务规则，将发现必须编写一个这样的方法，即接受一个输入并生成一个响应。这个方法必须是自给自足的——看起来不需要任何类实例变量和方法。有鉴于此，我们将编写一个静态的类方法。在编写单元测试期间，我们就能判断这样的选择是否是糟糕的。如果是糟糕的，我们总是可对其进行修改。



TDD的一大优点是，允许你改变错误的设计决策。

下面来创建类chapter03.backend.CharacterCounter。

(1) 在Package Explorer中，右击条目src/main/java并选择New>Class...。
(2) 在文本框Package中，输入chapter03.backend。
(3) 在文本框Name中，输入CharacterCounter。
(4) 确保选择了限定符public。
(5) 单击Finish按钮生成这个类，如下图所示。

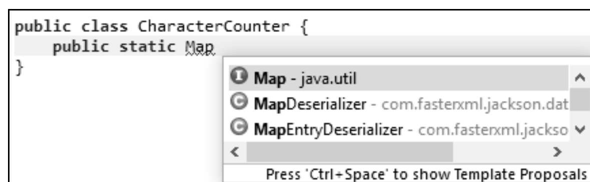


在上述屏幕截图中，调整了窗口的大小；在你看到的窗口中，显示的选项更多。向导生成的类如下：

```
package chapter03.backend;
public class CharacterCounter {
}
```

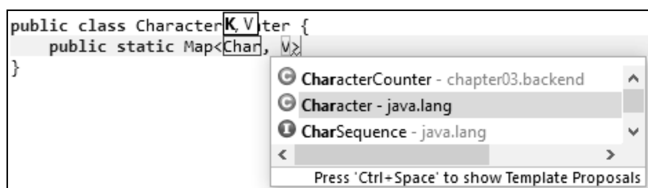
下面来编写这个方法的哑实现。将光标放到这个类中，并执行如下操作。

(1) 按Tab键缩进一层。
(2) 输入必不可少的访问限定符和非访问限定符（public static），再添加一个空格。
(3) 输入单词Map，再按住Ctrl和空格键。之后将出现一个窗口，其中显示了几个与你输入的名称匹配的类。选择Map - java.util并按回车，如下图所示：

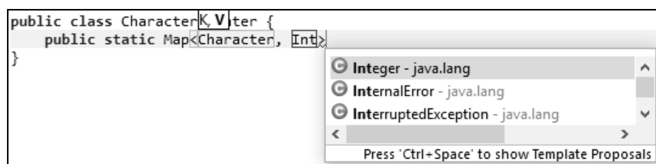


Eclipse将编写相应的导入语句和`Map<K, V>`，并将光标放在表示映射键`K`上。

(4) 输入`Char`并再次按住Ctrl和空格键，再选择Character - java.lang并按回车，如下图所示：



(5) 按Tab键移到字符`V`上，输入`Int`并按住Ctrl和空格键，再选择Integer - java.lang并按回车。这指定了映射中值的类型。

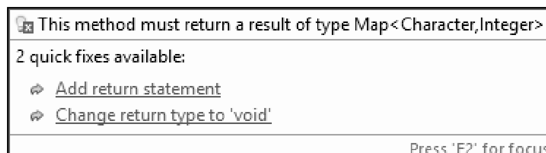


(6) 将光标移到行尾并输入`countCharacters(String text)`，再在当前行输入`{`，并在下一行输入`}`。

现在代码应类似于下面这样：

```
package chapter03.backend;
import java.util.Map;
public class CharacterCounter {
  public static Map<Character, Integer> countCharacters(String text) {
  }
}
```

Eclipse IDE指出这个类无法通过编译。将鼠标指向带红色下划线的方法名或左边的红色X图标，将出现一个工具提示，其中包含内容“This method must return a result of type Map<Character, Integer>”。Eclipse提供了两种自动解决方案：Add return statement（添加return语句）和Change return type to 'void'（将返回类型改为void），如下图所示：



这里选择第一种解决方案；为此，选择Add return statement。Eclipse将自动编写如下代码：

```
return null;
```

至此，我们编写了一个能够通过编译的后端类。下面来编写一个单元测试，以便检查我们的API是否正确。如果对结果满意，就可编写有效的实现，并检查它是否像预期的那样工作。

3. 创建测试用例类并编写其第一个单元测试

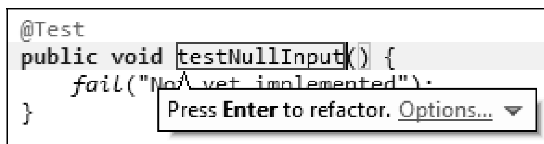
前面的业务规则明确地指出，如果传入的是null而不是String实例，这个类将引发`IllegalArgumentException`异常。下面来测试传入null时这个类是否会引发这种异常。为此，先来创建一个供测试框架JUnit使用的类（所有的测试都将包含在这个类中）。

(1) 在Eclipse IDE中，右击条目src/test/java并选择New>JUnit Test Case。
(2) 在文本框Package中，输入`chapter03.backend`。
(3) 在文本框Name中，输入`CharacterCounterTests`。
(4) 单击Finish按钮生成这个类。

生成的类如下（出于简洁考虑删除了一些空行）：

```
package chapter03.backend;
import static org.junit.Assert.*;
import org.junit.Test;
public class CharacterCounterTests {
  @Test
  public void test() {
    fail("Not yet implemented");
  }
}
```

注解`@Test`告诉框架JUnit，这是一个包含单元测试的方法。稍后你将看到，它可包含可选参数。包含单元测试的方法不能返回任何值，因此使用了关键字`void`来指定其返回类型。下面来重命名方法`test()`：将光标放在第一个字符上，再按Alt + Shift + R，将方法名改为`testNullInput`并按回车。



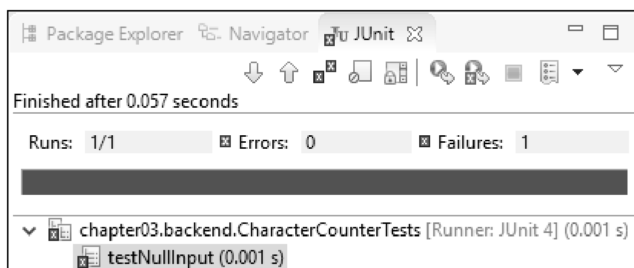


在这里，按Alt+Shift+R好像有点傻。但如果这个方法位于大型程序中，这样做将把修改应用于整个程序，因此务必尽可能这样做。用得越多，越容易记住键盘快捷键。

将光标放在`fail`语句所在的行，并按Ctrl + D将整行删除。将整个方法修改成下面这样（别忘了按Ctrl和空格键来输入`expected`、`IllegalArgumentException`和`CharacterCounter`，以减少键击次数）：

```
@Test(expected=IllegalArgumentException.class)
public void testNullInput() {
  CharacterCounter.countCharacters(null);
}
```

这些代码告诉JUnit，仅当引发了`IllegalArgumentException`异常时，这个测试才是成功的。如果类没有引发这个异常，这个测试就被视为失败的。现在正是运行这个测试的大好时机。为此按F11，Gradle将构建和编译代码并运行测试。



测试失败了，这符合预期。现在切换到选项卡Package Explorer，以便修复这个测试。

4. 实现输入有效性检查

打开src/main/java中的类`CharacterCounter`。在这个类的方法`countCharacters`中，在代码行`return null`上方输入如下代码：

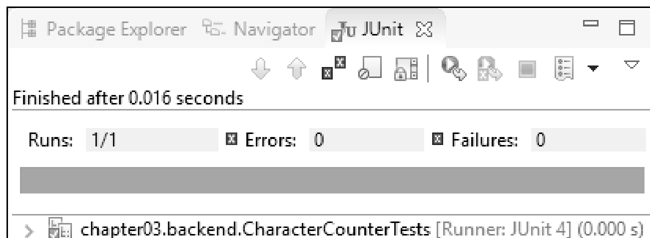
```
if (text == null) {
  throw new IllegalArgumentException("text must not be null");
}
```

这些代码的含义几乎是不言自明的。由于`IllegalArgumentException`是`RuntimeException`的子类，因此这个方法无需向编译器指出它可能引发这种异常。



为验证这一点，可双击类名`IllegalArgumentException`，再按F4，也可右击它并选择Open Type Hierarchy。之后，请单击标签Package Explorer返回到包资源管理器。

现在按F11运行这个测试，如下图所示：



这次测试成功了。祝贺你！高兴之余别忘了再次单击标签Package Explorer。

5. 编写第二个单元测试

下面再创建一个测试，对主业务逻辑进行测试。为添加新的单元测试，请打开src/test/java中的CharacterCounterTests类，在方法testNullInput()后面添加一个空行，再添加如下方法：

```
@Test
public void testStringInput() {
  Map<Character, Integer> map = CharacterCounter
                              .countCharacters("!a!A!");
  assertEquals(map.size(), 3);
  assertEquals(map.get('a').intValue(), 1);
  assertEquals(map.get('!').intValue(), 3);
  assertEquals(map.get('A').intValue(), 1);
}
```

我们传入一个字符串，并核实返回的映射包含的字符数未超过预期；另外，我们还测试了每个字符出现的次数。下面是一些注意事项。

- Character类是一个基本类型包装类，包装了基本类型char值。在Java中，将char值（单个UTF-16 Unicode字符）放在单引号中，而将String值放在双引号中。如果我们传入String "A"而不是字符'A'，映射的get方法将返回null。
- 映射的方法get返回包装类Integer的一个实例，其中包含指定字符出现的次数。我们将它转换为基本类型值，以便能够轻松地使用JUnit方法assertEquals的重载版本assertEquals(int, int)。这个方法有很多重载版本，如果我们不使用前面说的版本，就必须执行一些类型转换操作，以遵循Java的重载规则。

按F11，你将看到这个测试因异常NullPointerException而失败。之所以会引发这种异常，是因为返回的映射为null，导致调用map.size()失败。

6. 实现业务逻辑

对API感到满意后，我们来实现业务逻辑，以便让刚才的测试通过。切换到Package Explorer，打开src/main/java中的CharacterCounter类，并在检查输入是否为null的语句后面添加如下代码：

```
Map<Character, Integer> map = new HashMap<>();
for (char c: text.toCharArray()) {
  if (!map.containsKey(c)) {
    map.put(c, 1);
  } else {
    int curValue = map.get(c);
    map.put(c, ++curValue);
  }
}
return map;
```

对上述代码说明如下。

- 变量map指向一个HashMap实例，后者将一个Character实例（映射的键）映射到一个Integer实例（映射的值）。
- 数据类型String没有实现接口Iterable，无法用于改进的for循环中。我们必须调用其方法toCharArray()来返回一个char数组，而数组总是可用于改进的for循环中。
- 我们在这些代码中使用了基本类型char值。当这些char值被用于映射的方法中时，Java将自动把它们为Character实例，这都是拜自动装箱功能所赐。别忘了，泛型要求将类作为参数。
- 如果在映射中没有找到当前字符，就将其加入映射中，并将键设置为当前字符，而值设置为1。
- 如果在映射中找到了当前字符，就从映射中获取其当前值。
- 请注意代码行map.put(c, ++curValue);。它首先将值加1，再将键和修改后的值存储到映射中。作为练习，请读者尝试将++curValue改为curValue++，看看测试是否会失败。

按F11再次运行这两个测试，现在它们都应该成功。

7. 创建可执行的应用程序任务

这里不会使用SparkJava的单元测试功能，而创建一个Gradle能够运行的可执行程序。为此，最简单的方法是使用Gradle的插件application在这个Gradle项目中添加一个run任务。下面先来创建启动Web服务的类。

在Package Explorer中，添加一个名为Webservice的类，并将其放在chapter03.main包中。为此，可右击src/main/java并选择New>Class。在这个类中，添加向控制台打印简单字符串的方法main()，此时这个类应类似于下面这样：

```
package chapter03.main;
public class WebService {
  public static void main(String[] args) {
    System.out.println("The program is running!");
  }
}
```

使用Package Explorer打开文件build.gradle，并在apply plugin:'java'下方新增一行内容，如下所示：

```
apply plugin: 'java'
apply plugin: 'application'
```

添加一个空行，再添加如下内容：

```
mainClassName = "chapter03.main.WebService"
```

4

这个条目使用全限定名指定了插件application将运行的包含方法`main()`的类。在选项卡Gradle Tasks中，双击build > build来构建这个项目。如果一切顺利，请单击选项卡Gradle Tasks中右侧的第四个图标，其工具提示为“Refresh Tasks for All Projects”。这将添加插件application在这个Gradle项目中新增的任务。



现在可以在选项卡Gradle Tasks中选择任务application>run了。如果必要，将自动编译项目。为查看控制台输出，必须切换到选项卡Console。如果你不希望Eclipse自动切换到选项卡Gradle Execution，可禁用这项功能，为此需执行下面的步骤。

(1) 切换到选项卡Gradle Tasks。
(2) 展开条目JavaWebservice>application，右击其中的条目run并选择Open Gradle Run Configuration...。
(3) 取消选中复选框Show Execution View，再单击OK按钮关闭打开的对话框。

现在当你选择任务application>run时，将自动切换到选项卡Console，你将能够看到方法main打印到控制台的消息。

8. 创建Web服务

下面来将方法`main()`转换为一个这样的程序，即设置一条Spark路由以处理HTTP GET请求。为此，打开`chapter03.main.Webservice`类，并在`package`语句下方添加如下`import`语句：

```
package chapter03.main;
import java.util.Map;
import spark.Spark;
import com.fasterxml.jackson.databind.ObjectMapper;
import chapter03.backend.CharacterCounter;
```

在`WebService`类中，添加`private static`变量`mapper`，它是一个`ObjectMapper`实例。这个类来自Jackson库，将负责对方法`countCharacters`的输出（一个`Map<Character, Integer>`实例）进行转换。换而言之，这个`Map`对象将把从`Character`键映射到`Integer`值的`Map`实例映射到JSON：

```
private static ObjectMapper mapper = new ObjectMapper();
```

现在可以编写方法`main()`了。为此可修改原来的`main()`方法，但别忘了在编写代码时使用前面提到的组合键：

```
public static void main(String[] args) {
  Spark.get("/main", (req, res) -> {
```

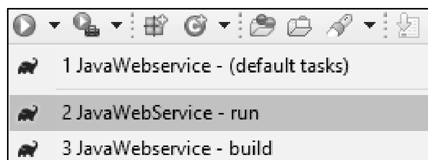
```
    res.type("application/json");
    try {
      String value = req.queryMap("value").value();
      value = (value == null ? "" : value);
      Map<Character, Integer> map = CharacterCounter
                              .countCharacters(value);
      return mapper.writeValueAsString(map);
    } catch (Exception e) {
      e.printStackTrace();
      return "{}";
    }
  });
}
```

对这些代码说明如下。

- 通过调用方法`Spark.get`，Spark库设置了一个处理程序，这个处理程序将在用户通过HTTP GET请求指定的URL时做出响应。定义HTTP处理程序后，Spark将确保HTTP服务器在程序初始化完毕后自动启动。
- 方法`Spark.get()`的第二个参数是一个lambda表达式。这里的lambda表达式接受两个参数：请求和响应。这两个参数都将被传入的lambdab表达式使用：请求（`req`）被用来读取HTTP请求的查询参数，而响应（`res`）被用来将Web服务的输出格式设置为JSON。这个lambda表达式将在用户通过HTTP GET请求URL /main时执行。
- 这里使用了一个以前没见过的“if”条件，其中`?`前面的部分为表达式。如果这个表达式的结果为true，就返回`?`后面的第一部分（这里为`""`），否则就返回`:`后面的那部分。在这里，如果从查询参数中获取的值为null，就将其改为空字符串，否则返回查询参数的值。
- 静态变量`mapper`是一个Jackson类`ObjectMapper`的实例，负责将`Map<Character, Integer>` map（将`Character`键映射到`Integer`值的`Map`）转换为有效的JSON表示。如果引发了异常，将向控制台打印错误消息，并返回一个空的JSON对象。

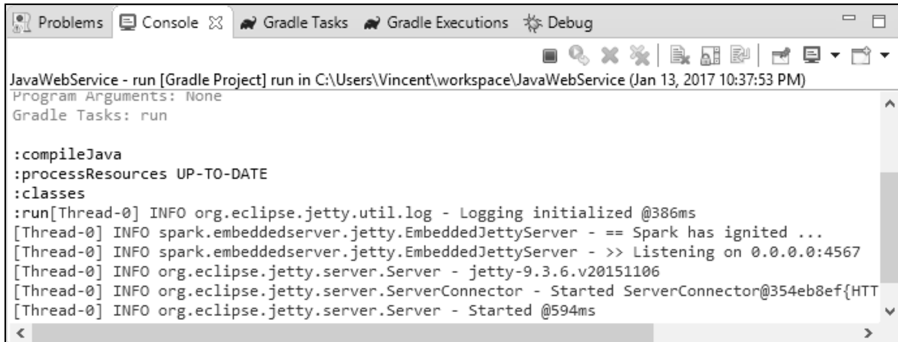
9. 运行这个Web服务

要通过Eclipse IDE顶部的工具栏中的绿色run图标运行这个Web服务，请单击这个图标旁边的箭头：



这里列出了通过这个按钮可执行的所有Gradle任务，包括选项build和run，它们分别启动构建过程和程序。请选择选项run，这样当你按Ctrl + F11或单击run图标时，Eclipse将启动应用程序。Eclipse IDE将一刻不耽误，马上启动应用程序。

过段时间后，应用程序将开始向选项卡Console中打印输出，其中的最后一条消息应包含Started以及SparkJava 在内部使用的服务器的类名，即`org.eclipse.jetty.server.Server`：



SparkJava使用的HTTP服务器默认使用端口4567。请启动浏览器，并访问http://localhost:4567/main?value=Test。

得到的输出应类似于下面这样：



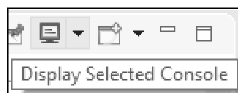


输出的顺序可能与这里显示的不同，原因是`HashMap`类不以任何有意义的方式排列元素。

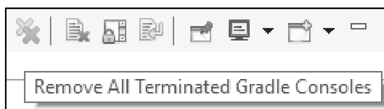
要停止这个应用程序，可单击Console选项卡中红色的停止按钮。



HTTP服务器可能需要过段时间才会停止，但最终这个按钮将变成灰色，而应用程序本身将随之停止。每个Gradle操作都会打开一个新的控制台。要在这些控制台之间切换，可单击工具提示为“Display Selected Console”的按钮，也可单击这个按钮右边的箭头，这将列出所有打开的控制台：



要关闭所有非活动窗口，可单击工具提示为“Remove All Terminated Gradle Consoles”的按钮。在你在开发项目期间启动了大量Gradle任务时，这个按钮提供了极大的方便：



作为练习，在`CharacterCounter`类中，尝试将`java.util.HashMap`实例替换为`java.util.TreeMap`实例。`TreeMap`类会根据插入的键排列元素，同时由于它像`HashMap`类一样实现了接口`Map<K, V>`，因此这样替换后程序依然能够正确运行。在实际工作中，通过使用接口来隐藏实现细节确实是个很好的主意。

10. 创建Javadoc文档

现在是创建一些文档的绝佳时机。在选项卡Gradle Tasks中，单击任务Documentation>Javadoc。你必须在Package Explorer中刷新项目。然后，打开选项卡Navigator，并选择项目的构建文件——build>docs>javadocs>index.html，再右击它并选择Open With>System Editor，这将打开默认浏览器。

要给类或方法添加文档，可在类或方法的定义前面输入`/**`并按回车。

在Java中，常规多行注释放在`/*`和`*/`之间（与C和C++一样），而单行注释以`//`打头。Javadoc注释以`/**`打头，并以`*/`结尾（与常规多行注释一样）。Eclipse会自动创建一个分别以`/**`和`*/`打头和结束的块，并添加一些属性。文档必须由你来提供，同时别忘了你需要编写HTML，因此在使用某些字符时必须进行转义，例如，使用`>`来表示`>`。

4.3　小结

很多人认为，开发Web应用程序时，使用Java要比使用其他现代语言编写多得多的代码，但愿我们消除了这种误解。编写代码时，我们使用了Eclipse IDE提供的各种功能来简化开发工作并提高其速度。我们使用了Gradle来管理依赖和构建项目，我们还添加了Gradle插件application，以便能够轻松地运行应用程序——只需运行一个简单的Gradle任务。通过使用TDD，我们编写了一个后端类。为将这个类的输出转换为JSON，我们使用了Jackson库，还使用了SparkJava框架来创建基于这个类的Web服务。

如果你对SparkJava框架感兴趣，请务必访问http://sparkjava.com。

下一章将介绍Scala，它既提供了强大的函数式编程支持，又是一种纯粹的OOP语言。

第 5 章 Scala

Scala很独特，既提供了强大的函数式编程支持，又是一种纯粹的面向对象编程（OOP）语言。本章将介绍Scala的这两个方面。

Scala提供了两种运行代码的方式。它提供了一个交互式shell，让你能够直接输入代码并马上运行它们；这个程序还可用来直接运行Scala源代码——无需先手动编译。Scala还提供了传统编译器`scalac`，这种编译器将Scala源代码编译成Java字节码，并生成扩展名为.class的文件。本章只介绍第一种方法，编译器`scalac`将在下一章介绍。

Scala自带了标准库，作为Java运行时环境（JRE）中随Java开发包（JDK）一起安装的Java类库的补充。Scala标准库包含众多为方便使用Scala功能而进行了优化的类，如与Java集合兼容的集合类。

本章将讨论如下主题：

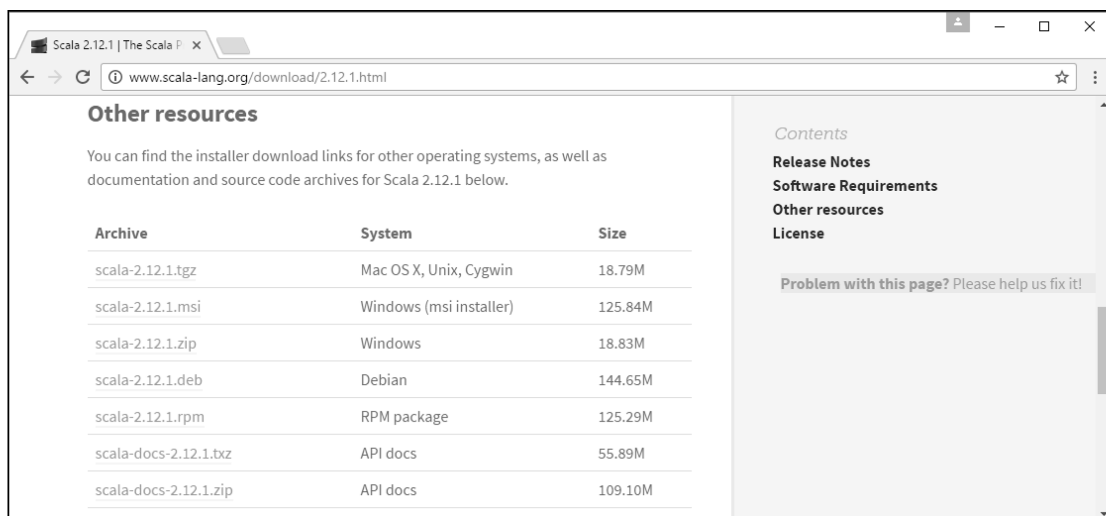
- 安装Scala；
- Scala的读取–评估–打印–循环shell；
- 函数式编程和命令式编程；
- Scala语法和规则；
- Scala的OOP功能；
- Scala标准库；
- Scala的函数式编程功能。



本章使用的很多概念都在第3章介绍过，因此建议你先阅读第3章，再阅读本章。

5.1 安装 Scala

可从Scala官网下载最新版本的Scala：



Scala提供了用于众多不同操作系统的版本。你可下载归档文件（对于Windows操作系统，为ZIP文件；对于Linux和macOS操作系统，为.tgz文件）并手动安装，但对于流行的操作系统，提供了自动安装包。

在DOWNLOAD页面中，向下滚动到Other resources部分，找到所需的归档格式并下载。你只需将归档文件解压缩，并将其子目录bin添加到环境变量Path即可。有关在Windows、macOS和Linux系统中如何将目录添加到环境变量Path中，请参阅第2章。如果你选择使用安装程序进行安装，只需按提示操作即可。

要核实是否正确地安装了Scala，只需打开命令提示符（Windows）或终端（Linux或OS X），再输入如下命令并按回车：

```
scala
```

如果安装成功，控制台将出现类似于下面的输出[①]：

```
Welcome to Scala 2.12.1 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_112).
Type in expressions for evaluation. Or try :help.

scala>
```

要退出Scala，可输入:quit并按回车。

Scala网站提供了在线文档；为方便参考在线文档，推荐你将如下URL加入书签：

① 默认都是安装在Program Files (x86)或Program Files，但这两个目录都包含Scala当前不支持的空格，因此你需要指定一个不包含空格的安装目录，否则无法运行scala命令。——译者注

- http://docs.scala-lang.org
- http://www.scala-lang.org/api/current/

5.2 Scala 的 REPL shell

前一节提到的命令`scala`会启动Scala交互式shell，即Scala的**读取–评估–打印–循环**（Read-Eval-Print-Loop，REPL）环境。你输入代码行后，REPL程序将评估它，并在合适的情况下打印响应。这个程序将不断地这样做，直到你退出为止。

```
Command Prompt - scala

C:\Java\Languages\scala-2.12.1\bin>scala
Welcome to Scala 2.12.1 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_112).
Type in expressions for evaluation. Or try :help.

scala>
```

在Scala shell中，你可以交互的方式编写Scala代码。鉴于Scala是一种编译型（而不是解释型）语言，你可在这个程序中动态地输入并执行Scala代码，但在幕后，Scala将编译你输入的代码并运行编译后的版本。Scala交互式shell是为尝试Scala表达式而设计的，并不适用于编写完整的程序，但非常适合用于尝试本章的代码片段。这个shell提供了自己的命令；要了解所有的shell命令，可输入命令`:help`并按回车。

本章只使用命令`scala`来运行Scala代码，至于编译器`scalac`，将在下一章介绍。要运行本章的代码，可在shell中直接输入它们，也可使用文本编辑器创建一个包含源代码的文件，再将该文件的路径作为参数传递给命令`scala`来运行脚本。这样shell将编译指定的脚本，并立即运行它，但不保存编译得到的文件。另外，脚本运行完毕后，shell将自动退出。

5.3 函数式编程和命令式编程

从本质上说，Java是一种命令式编程语言。在命令式编程语言中，变量通常是可修改的，而类通常保存了内部状态。在Java中，POJO（Plain Old Java Object）是命令式编程的典范。标准POJO包含可通过调用设置方法随便修改的变量，可访问POJO实例的任何代码都可修改其变量。这可能导致难以发现的微妙bug，多个线程试图同时修改同一个变量时尤其如此。

在函数式编程中是这样编写代码的，即确保在程序运行期间不会修改任何既有的变量。值是

以函数参数的方式指定的，而输出是根据参数生成的。每次调用函数时，只要指定的参数相同，输出就必须相同。

下面来看一个非常简单的示例。请不要过多地关注其中的语法，因为本章后面将详尽地介绍Scala语言的语法。先来看一个传统的Scala面向对象编程示例：

```
class AddDemoOOP {
  var x = 0
  def add(y: Int): Int = {
    x += y
    x
  }
}

val a = new AddDemoOOP()
print(a.add(1))
print(a.add(1))
```

这将打印1和2。虽然两次调用时方法add接受的参数相同（都是整数1），但它们返回的值不同。这是因为这个方法修改了类的状态。在纯粹的函数式编程中，这是不允许的。下面是这个类的函数式版本：

```
class AddDemoFunctional {
  def add(x: Int, y: Int): Int = {
    x + y
  }
}

val b = new AddDemoFunctional
print(b.add(0, 1))
print(b.add(0, 1))
```

这将打印1两次。这个版本的类没有存储任何内部状态。要让方法add返回不同的值，必须给它传递不同的参数。



注意，Scala虽然是纯粹的OOP语言，但并不是纯粹的函数式编程语言。如第一个示例所示，使用Scala很容易编写不遵循函数式编程规则的代码。

编写使用多个线程的程序时，函数式编程是一种流行的选择。由于方法不能修改在多个线程中使用的数据结构的状态，因此函数式编程通常比命令式编程安全得多，但这要求开发人员使用不同的思维方式。

有关函数式编程可说的还有很多，本章后面将介绍其他一些与此相关的主题。



不同于众多其他的函数式编程语言，Scala让你能够按自己的步伐学习函数式编程，因为它是纯粹的面向对象编程语言。

5

5.4　Scala 语法和规则

Scala不像Java那样严格而繁琐：分号是可选的；在不必要的情况下，函数调用中的小括号也并非必不可少的（请看前述示例中的代码行`val b = new AddDemoFunctional`）。本节介绍如下主题：

- 静态类型语言；
- 可修改的变量和不可修改的变量；
- 常用的Scala类型。

5.4.1　静态类型语言

与Java一样，Scala也是一种静态类型语言：使用变量前必须先声明。Scala也是一种强类型语言：你总是可以指定要使用的类型，就像在Java代码中一样；但与Java不同的是，并非必须显式地指定类型。

声明方法的输入参数和返回值时，必须指定类型，但在方法或函数中声明变量时，并非必须指定类型，因为Scala编译器通常能够根据代码推断出变量的正确类型。

下面是一个这样的示例：

```
var i = 10;
var j = new java.lang.Object();
```

声明可修改的变量后，就可使用它来存储指定类型或可向上转换为该类型的值。例如，可在上述代码后面编写如下代码：

```
j = "Hello world"
```

由于`String`类型可向上转换为`java.lang.Object`，因此可使用变量`j`来存储指向字符串`"Hello world"`的引用。但不能将`String`赋给变量`i`，因为你将一个`Int`实例赋给了这个变量，`String`不能向上转换为`Int`实例。

可显式地指定变量的类型；使用类系列时，可能必须提供这种信息：

```
val i: Integer = 10
```

5.4.2　可修改的变量和不可修改的变量

Scala支持两种变量。声明方法的参数或类的实例成员时，必须使用下面两个关键字之一。

- `var`：用于声明可修改的变量；
- `val`：用于声明固定变量。

可修改的变量相当于Java中的普通变量，它们是可修改的，可随便修改；固定变量相当于Java中的`final`变量，只能给它们赋值一次。如果固定变量指向一个可修改的类实例，你依然可以修改这个实例的内容，这在第2章讨论过。



进行函数式编程时，应尽可能使用不可修改的变量。不可修改的值是函数式编程的基石。

Scala不支持静态变量（也叫类变量），但正如后面将介绍的，它支持可用于替代静态变量的单例对象。

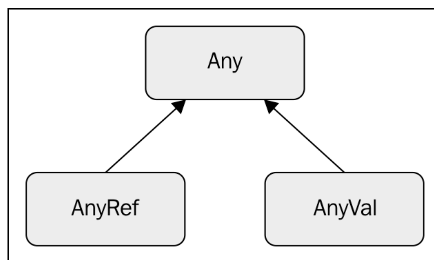
5.4.3　常用的 Scala 类型

在Scala中，可使用常见的Java类，但Scala也提供了自己的类，你应尽可能使用这些类。这些类的工作原理带有Scala的烙印，这里介绍其中的如下几个：

- `Any`；
- `AnyRef`；
- `AnyVal`；
- 字符串。

1. Any类

在Java中，祖先类为`Object`，而在Scala中，祖先类为`Any`，因此在没有显式指定时，所有的类都隐式地继承`Any`类。



如上图所示，`Any`类有两个子类：

- `AnyRef`；
- `AnyVal`。

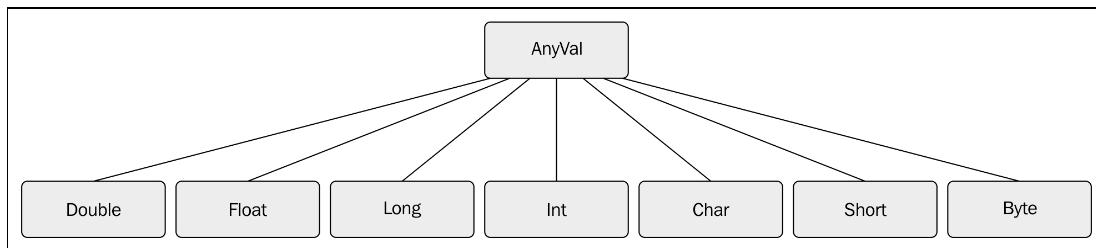
(1) `AnyRef`类——引用类

`AnyRef`类由引用变量使用，类似于Java类`java.lang.Object`，提供的方法也类似，如

equals()、hashCode()和finalize()。Scala中的大部分类都直接或间接地继承了AnyRef类。

(2) AnyVal类——值类

不同于Java，Scala是一种纯粹的面向对象语言，因此不支持基本类型值，而是将它们封装在包装类中：



这些包装类都是AnyVal的子类，它们被称为**值类**，Scala编译器以特殊的方式处理它们。

你可能会问，Scala为何使用自己的包装类，而不使用Java类库中的包装类。一个原因是Scala力图通过避免不必要的装箱来改善性能，为此需要有自己的内部逻辑；另一个重要的原因是Scala支持运算符重载（这不同于Java）。Scala包装类实现了用于计算的所有二元运算符，稍后将更详细地讨论这一点。

2. 字符串

很多JVM语言都提供了自己的字符串类，这些类除了Java标准类String的方法和字段外，还提供了其他的便利方法和字段，但Scala不是这样的，它使用Java类库中java.lang包中的String类。

Java字符串是不可修改的。如果方法要修改字符串，它将返回一个新的String实例，其中包含修改后的字符串，而原来的String实例保持不变。这种不变性正是Scala的函数式编程功能所需要的。

5.5 Scala 的 OOP 功能

与Java一样，Scala编译器也要求将代码封装在类中。为了满足这种要求，Scala的交互式REPL shell自动将你输入的代码封装在一个在幕后生成的不可见的类中。执行命令scala后，你可以立即开始编写要执行的函数或代码，就像使用Python或其他脚本语言时那样。本节介绍如下主题：

- 定义包和子包；
- 导入成员；
- 定义类；

- 实例方法和实例变量；
- 构造函数；
- 扩展类；
- 重载方法；
- 抽象类；
- 特质；
- 单例对象；
- 运算符重载；
- Case类。

5.5.1　定义包和子包

Scala提供了package语句，你可以在文件开头使用它：

```
package PACKAGENAME
```

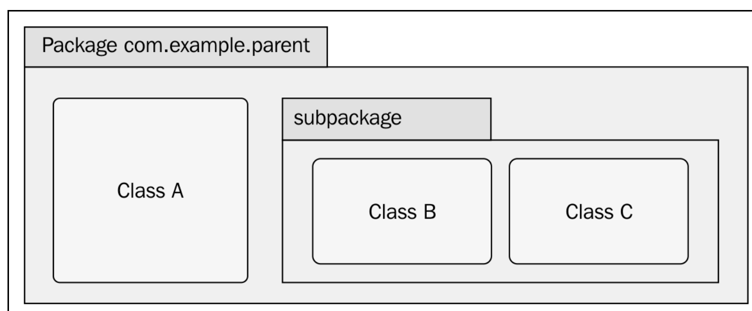
这种语句的工作原理与Java中完全相同：在前述文件中定义的每个类都将放在PACKAGENAME包中。

然而，Scala让你有更大的控制权。稍后你将看到，Scala还支持子包：在Scala中，前缀相同的包将自动关联起来。在源代码文件中，可使用package语句来定义子包：

```
package com.example.parent
class A {
}

package subpackage {
  class B { }
  class C { }
}
```

上述代码创建了三个公有类，其中类A位于com.example.parent包中，而类B和C位于com.example.parent.subpackage包中：





在REPL shell中，不能使用package语句，这是因为这个shell将生成一个不可见的类，其中包含你输入的代码。如果源代码包含package语句，就必须使用编译器scalac来编译它。

5.5.2　导入成员

在Scala中，import语句的功能比在Java中更强大，但其基本形式与Java中一样：

```
import com.example.parent.A
```

为导入包的所有成员，Scala不使用通配符*，而使用_：

```
import com.example.parent._
```

可在同一条语句中导入多个成员：

```
import com.example.parent.subpackage.{B, C}
```

也可将导入的成员重命名：

```
import com.example.parent.subpackage.{C => D}
```

在前面的示例中，在源代码中引用com.example.parent.subpackage.C类时，必须使用类名D。这在名称发生冲突（不同的成员同名）时提供了极大的便利。

前面说过，Scala支持子包，而子包可访问其父包的私有成员。在前面的示例中，这意味着com.example.parent.subpackage中的代码能够访问com.example.parent的私有成员——甚至都不需要导入这些类。

另一个很不错的功能是可导入包：

```
import com.example.parent
```

有了上述代码后，就可在代码中引用parent包的成员：

```
var c = new parent.subpackage.C()
```

5.5.3　定义类

从前面的一些示例可知，在Scala中定义类的方式与在Java中很像：

```
class TheClassName {
}
```

在Scala中，可在一个源代码文件中定义任意数量的公有类。源代码文件的名称不必与其定义的任何类相同。

对于类，Scala支持如下显式的访问限定符：

- private

在没有指定访问限定符的情况下，类默认为公有的，这与Java差别很大。在Scala中，不能创建包私有的类，也没有显式的访问限定符`public`。如果要创建一个空类，可不指定代码块`{ }`。

5.5.4 实例变量和实例方法

类可包含实例变量和实例方法。

Scala不支持静态成员（类变量和类方法），因此没有`static`或与之等价的关键字。为填补这种空白，Scala支持单例类，这将在本章后面介绍。

1. 实例变量

要添加实例变量，只需在类中定义`def`和`val`变量即可。在大多数情况下，可不显式地指定变量的类型，因此Scala将根据初始值推断变量的类型，但如果你愿意，也可以显式地指定：

```
var anIntegerVariable: Int
val anIntegerValue = 0
```

在显式地指定了类型的情况下，可不立即显式地对实例变量进行初始化，在这种情况下，它将被自动初始化为空值。没有指定类型时，必须在声明变量的同时给它赋值，否则Scala将不知道变量是哪种类型。

2. 实例方法

在Scala中，方法声明与Java中很像。如果方法有输入参数，必须指定其类型，另外，方法可什么都不返回，也可返回一个对象实例。对于返回类型，可显式地指定，也可不指定：

```
def methodName(parameter1: Int, parameter2: Int): Int = {
  parameter1 + parameter2
}
```

与Java不同的一个点是，可不显式地指定返回类型——除非Scala编译器认为存在二义性。在前面的示例中，编译器知道返回值为两个`Int`实例的和，因此返回类型必然是`Int`。因此，将其修改成下面这样也能通过编译：

```
def methodName(parameter1: Int, parameter2: Int) = {
  parameter1 + parameter2
}
```

在任何方法中，都将自动返回最后一个表达式的值。Scala提供了显式的`return`语句，但并非必须使用它，同时推荐不使用它。在Scala中，方法和函数不能提早结束。使用了`return`语句

时，必须显式地指定方法的返回类型。

在Scala中，如果方法什么都没有返回（相当于Java中的void），可将返回类型指定为类名Unit：

```
def methodWithoutReturnValue(): Unit = {
}
```

如果没有指定返回类型，且方法中没有只包含一个表达式的代码行，Scala将认为返回类型为Unit：

```
def methodWithoutReturnValue() = {
}
```

如果方法显式地将返回类型指定为Unit，同时又包含只有一个表达式的代码行，编译器将发出警告，同时忽略这个表达式。

如果方法的实现只包含一行代码，则可省略表示代码块的{ }，因此下面的代码是合法的：

```
def helloWorld() = println("Hello world")
```

3. 用于实例成员的访问限定符

与类一样，在没有显式指定访问限定符的情况下，类成员也是公有的。Scala支持的其他访问限定符如下：

- protected
- private

在访问限定符方面，Scala和Java存在一些重要的差别。

- Scala支持子包的概念。子包中的类可访问其父包的私有成员，这在前面介绍import语句时讨论过。
- 在Scala类中，使用访问限定符protected的成员对当前包中的其他类来说是不可见的，而在Java中，受保护的成员对当前包中的其他类来说是可见的。

5.5.5　构造函数

主构造函数是通过在类代码块中添加参数和输入类型定义的：

```
class ClassWithParameterizedConstructor(var parm1: Int, parm2: Int)
{
  println("This code is executed as part of the constructor")
}
```

这定义了一个主构造函数以及实例变量parm1和parm2（这两个变量的类型都是Int）。这里

有几点需要注意。

- Scala自动创建与参数同名的字段。
- 对于`var`字段，将自动为其生成公有的获取方法和设置方法，因此能够访问这个类的代码可随便访问构造函数的参数。
- 没有指定关键字`var`或`val`时，默认为`val`。
- 类的所有代码都可访问这些变量和值，嵌套的类、方法和函数亦如此。

主构造函数被调用时，将执行类中的语句。构造函数可重载；在Scala中，重载的构造函数被称为**辅助构造函数**（auxiliary constructor）。要创建其他的构造函数，可使用关键字`this`：

```
class ClassWithParameterizedConstructor(var parm1: Int,
                                        val parm2: Int) {
  def this(parm1: Int) = this(parm1, 0)
}
```

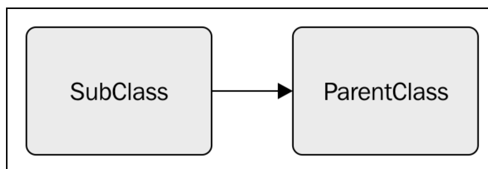
与方法一样，如果构造函数包含多行代码，可在等号后面使用`{ }`将这些代码括起：

```
class ClassWithParameterizedConstructor(var parm1: Int,
                                        val parm2: Int) {
  def this(parm1: Int) = {
    this(parm1, 0)
  }
}
```

对于辅助构造函数，有一条重要的规则：在辅助构造函数中，必须首先调用主构造函数或其他已定义的辅助构造函数。

5.5.6 扩展类

类是使用你熟悉的关键字`extends`来扩展的：




```
class ParentClass {
}

class SubClass extends ParentClass {
}
```

像Java一样，Scala也支持单继承。如果没有显式地继承任何类，将隐式地继承`AnyRef`类（`Any`的子类）。

在Scala中，只有子类的主构造函数能够调用父类的构造函数，这是像下面这样完成的：

```
class ParentClass(param1: Int, param2: Int) {
}

class SubClass(var param1: Int) extends ParentClass(param1, 10) {
}
```

SubClass的主构造函数（接受一个参数）调用ParentClass的主构造函数（接受两个参数）。如果ParentClass有辅助构造函数，也可以调用它们。

稍后你将看到，关键字extends也用于实现特质（trait）。通过扩展类和实现特质时，必须先指定要扩展的类。

重写方法

要在子类中重写方法，可使用关键字override：

```
class ParentClass {
  def test() = print("Hello, from the parent class")
}

class SubClass extends ParentClass {
  override def test() = {
    super.test()
    print(" and from the child class as well")
  }
}
```

正如这里演示的，要访问父类的成员，可使用关键字super。

5.5.7　重载方法

Scala支持方法重载，其中的工作原理和遵循的规则与Java中相同，下面是一个这样的示例：

```
class OverloadExample {
  def anOverloadedMethod(i: Int) { }
  def anOverloadedMethod(s: String) { }
}
```

5.5.8　抽象类

要创建抽象类，可在关键字class前面加上abstract：

```
abstract class AbstractClassName {
  def methodWithNoImplementationYet
  def methodWithImplementation() { }
}
```

不同于Java，抽象方法无需使用关键字`abstract`指定，而只需不给它提供实现。

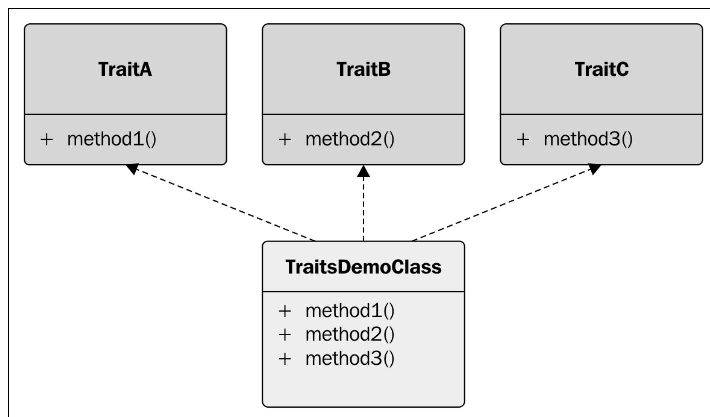
5.5.9 特质

特质很像Java接口。与Java 8接口一样，特质可定义抽象方法，也可定义包含实现的方法。下面来看一个示例：

```
trait TraitName {
  def methodWithImplementation() {
    // 代码在这里……
  }

  def methodWithoutImplementation()
}
```

Scala使用关键字`extends`来扩展父类，还使用它来实现特质。在Java中，类可以实现任意数量的接口；同样，在Scala中，类也可以扩展任意数量的特质。



有趣的是，`extends`列表中的条目是使用关键字`with`分隔的：

```
trait TraitA { def method1() }
trait TraitB { def method2() }
trait TraitC { def method3() }

class TraitsDemoClass extends TraitA with TraitB with TraitC {
  def method1() { }
  def method2() { }
  def method3() { }
}
```



注意，同时扩展一个类以及一个或多个特质时，必须先指定要扩展的类，否则将出现编译错误。

扩展一个或多个特质的抽象类可以给特质的抽象方法提供实现，但并非必须这样做。具体类必须提供所有特质的实现，这可以是直接提供的（如前面所示），也可以是间接提供的（如扩展其他提供了特质实现的类）。

5.5.10　单例对象

Scala提供了一种便利的对象。这种对象很像类定义，但不同之处在于，它不仅创建类，还创建一个可通过指定名称引用的对象实例。这就是单例类，即确保只创建其一个实例，如下例所示：

```
object SingletonObjectName {
  var x = 100
  def printX() = println(x)
}
```

这个对象不需要实例化，因为Scala将自动完成这种工作。要访问实例SingletonObjectName，只需使用其名称即可：

```
SingletonObjectName.x = 250
SingletonObjectName.printX()
```

这将打印250。当然，在单例对象中添加可修改的变量不是什么好主意，在它将被多个线程使用时尤其如此。任何时候，使用可修改的全局变量都是馊主意。

鉴于Scala不支持静态类成员，可将单例对象作为替代品。由于这种类在任何情况下都只有一个实例，因此使用它与将数据存储在静态变量中或定义静态方法的效果相同。

5.5.11　运算符重载

前面讨论AnyVal子类时说过，Scala支持运算符重载。在Java中，不能重写+和*等运算符，你只能将它们用于基本类型（将它们用于包装类时，将在内部将包装类拆箱为基本类型，执行完计算后再装箱为包装类）。

在Java中执行1 + 1时，Java编译器将二进制Java字节码编译成知道如何将两个整数值相加的JVM命令。不能将Java运算符用于自定义类，如果你对自定义类的实例调用+，编译器将拒绝编译代码。例如，下面的Java代码不能通过编译：

```
class A {
  public static void main(String[] args) {
    // 编译错误：二元运算符+的操作数的类型不正确
    A result = new A() + new A();
  }
}
```

而Scala以普通方法的方式实现了运算符。Scala类Int包含方法+，因此在Scala中，可在自定

义类中重写并实现运算符。如果你在自定义类中重写了运算符+，则当你在代码中使用运算符+将两个自定义类的实例（甚至两个不同类的实例）相加时，方法+将决定如何做。下面是一个简单的Scala自定义类，它实现了运算符+：

```
class CustomClass(var x: Int) {
  def + (other: CustomClass) = {
    new CustomClass(x + other.x)
  }
}

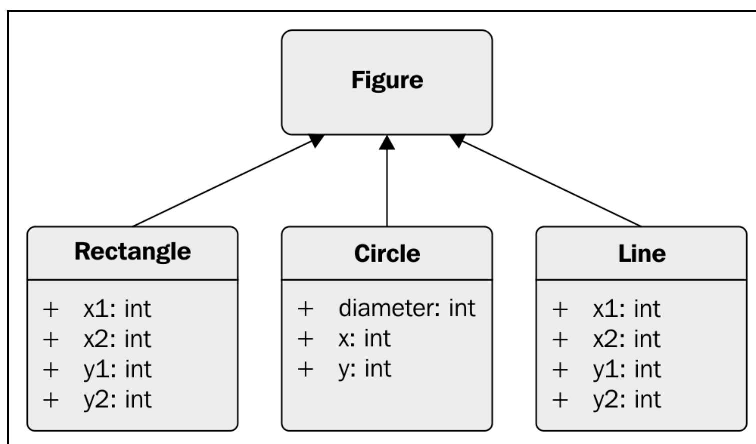
val result = new CustomClass(400) + new CustomClass(155)
print(result.x)
```

这将打印`555`。

5.5.12　Case 类

Scala支持一种特殊的类——`Case`类。作为程序员，你可能见过很像但又存在细微差别的数据结构。为处理这些数据结构，通常必须编写难以维护的`switch... case`（C、C#、Java和JavaScript）、`case...when`（Ruby）、`Select Case`（Visual BASIC）或`if ...else`块（其他语言），从中获取正确的数据并做相应的处理。

`Case`类为处理这种问题提供了优雅的途径，我们来看一个简单的示例：



`Case`类`Rectangle`、`Circle`和`Line`扩展了抽象类`Figure`：

```
abstract class Figure
case class Rectangle(x1: Int, y1: Int, x2: Int, y2: Int) extends
           Figure
case class Circle(x: Int, y: Int, diameter: Int) extends Figure
case class Line(x1: Int, y1: Int, x2: Int, y2: Int) extends Figure
```

首先，创建了空的抽象类Figure，以便能够将Case类进行逻辑分组。我们声明了一些Case类，它们表示绘图程序能够绘制的各种图形，其中每个Case类都包含相应图形所需的字段。创建这些类的实例易如反掌：

```
val rectangle = Rectangle(10, 20, 80, 50)
val circle = Circle(100, 200, 30)
```

有趣的是，实例化Case类时没有使用关键字new。要处理这些类，需要使用一种名为**模式匹配**的技术：

```
def drawFigure(figure: Figure): Unit = {
  figure match {
    case Rectangle(x1, y1, _, _) => _draw(x1, y1)
    case Circle(x, y, _) => _draw(x, y)
    case Line(x1, y1, _, _) => _draw(x1, y1)
  }

  def _draw(x: Int, y: Int): Unit = println("Start drawing at "
                                           + x + ", " + y)
}

drawFigure(rectangle)
drawFigure(circle)
```

根据约定，使用下划线（_）来表示当前不需要的字段。

5.6　Scala 标准库

详细讨论OOP后，下面来着手编写有点用的类和方法。随Scala安装了Scala标准库，这个库很大，包含Scala特有的类。本节讨论如下主题：

- 泛型；
- 集合；
- XML处理。

5.6.1　泛型

Java使用表示法ClassName<T>来表示支持泛型的类。本书前面说过，定义有些类（如接口Map<K, V>）时，需要指定多种类型。对于Map，需要指定它将存储的键和值的类型。

在Scala中，使用表示法ClassName[T]：

```
val aList = List[Int](1, 2, 3, 5)
```

这将创建一个不可修改的列表，其中包含5个元素。由于指定了List[Int]，因此只能在其

中添加类型为Int或可向上转换为Int的类的实例。

同样，对于需要指定两种类型的泛型类，可这样定义：

```
val m = Map[String, String]("key1" -> "value1", "key2" -> "value2")
```

这里创建了一个映射，它将键"key1"和"key2"分别映射到字符串值"value1"和"value2"。

5.6.2 集合

Scala标准库提供了很多集合API，这些API分为两大类：

- 不可修改的集合；
- 可修改的集合。

不可修改的集合通常位于scala.collection.immutable包中，而可修改的类位于scala.collection.mutable包中。默认自动导入的Scala包包含指向多个不可修改的集合类的引用，因此，如果没有显式地导入不可修改或可修改的集合类，默认使用的将是不可修改的版本。

1. 不可修改的列表

在前面两个示例中，使用的分别是不可修改的列表和映射。这些数据结构将在初始化期间被填充。可使用不可修改的集合来创建新列表，下面的示例基于既有列表的内容创建新列表：

```
val immutableList = List[Int](1, 2, 3, 4, 5)
val newImmutableList1 = 0 :: immutableList
val newImmutableList2 = immutableList ::: List(6, 7)
```

乍一看，这可能让人迷惑。第2行创建一个新的列表实例，其中包含0和另一个列表的副本，因此newInstance1为List[Int](0, 1, 2, 3, 4, 5)。运算符::左边是要添加到新建实例中的值，而右边是要复制的列表。由于List类针对LIFO（后进先出）操作进行了优化，因此创建新列表实例时，将新元素放在既有列表的元素前面会更容易。

然而，创建新列表时，也可在列表末尾添加新元素。为此可使用运算符:::，但这个运算符两边都必须是列表。在前面的示例中，newImmutableList2将包含如下元素：1、2、3、4、5、6、7。

请注意，创建新列表时，并没有修改原来的列表aList。

2. 可修改的列表

还有可修改的列表版本，这种列表名为ListBuffer，位于scala.collection.mutable包中。完全可以想见，这种版本提供了你熟悉的方法，如append()、remove()和clear()：

```
import scala.collection.mutable
val aMutableList = mutable.ListBuffer[Int](1, 2, 5)
aMutableList.remove(2)
aMutableList.append(3)
println(aMutableList(0))
println(aMutableList)
```

这将打印1和ListBuffer(1, 2, 3)。

在Scala中，一种最佳实践是尽可能使用运算符。下面是使用运算符实现的前一个示例：

```
import scala.collection.mutable

val aMutableList = mutable.ListBuffer[Int](1, 2, 5)
aMutableList -= 5
aMutableList += 3
println(aMutableList)
```

运算符-=将指定的值从列表中删除，而+=在列表中添加新值。



注意，使用运算符-=，必须指定要删除的值。另外，请注意将删除的元素的索引。

在Scala提供的众多集合类中，有一个名为ArrayBuffer的可修改类。这个类在内部是使用数组实现的，因此使用索引来访问其元素的效率要高得多：

```
import scala.collection.mutable

val aMutableArray = mutable.ArrayBuffer[Int](1, 2, 3)
aMutableArray += 4
println(aMutableArray(3))
```

这将打印4。与ListBuffer类一样，ArrayBuffer也实现了很多运算符。

3. 不可修改的映射

Scala标准库包含众多实现各不相同的映射类，这里将讨论标准类Map，它是不可修改的：

```
val immutableMap = Map[Int, String](10 -> "ten", 20 -> "twenty")
println(immutableMap(20))
```

基于既有的映射创建新的映射实例很容易，下面的示例修改了前面定义的映射：

```
val newImmutableMap = immutableMap + (30 -> "thirty")
```

要将两个映射合而为一，可使用运算符++：

```
val combinedMap = newImmutableMap ++ Map[Int, String]
                  (24 -> "twentyfour")
```

4. 可修改的映射

Scala标准库中的可修改映射类之一是`HashMap`，它很像Java类`java.lang.HashMap`：

```
import scala.collection.mutable

val mutableMap = mutable.HashMap[Int, String](10->"ten",
                                              50->"fifty")
mutableMap += (100 -> "Hundred", 150 -> "Hundred and fifty")
mutableMap -= 10
println(mutableMap)
```

这将打印`mutableMap.type = Map(50 -> fifty, 100 -> Hundred, 150 -> Hundred and fifty)`。



注意，在前面的不可修改映射示例中演示的运算符也可用于`HashMap`实例。用于`HashMap`实例时，这些运算符的工作原理不变，也将创建新实例，而不是修改当前`HashMap`实例。

5

5.6.3 XML 处理

Scala标准库包含一个功能强大的XML库，可帮助创建和使用XML文档。这里将演示如何使用XML字面量来生成XML文档。通过使用这种功能，可直接在Scala源代码中输入XML内容。在幕后，Scala编译器将使用其XML库来填充变量，并验证生成的XML是否有效。



建议你不要直接在Sacla REPL shell中输入下面的示例，而使用你喜欢的文本编辑器创建一个源代码文件，再将其路径传递给命令`scala`。直接在REPL shell中输入代码和XML时，这个交互式解析器可能出现故障。

下面是一个简单的示例，它生成一个包含XML的`String`：

```
val productCode = "PC Monitor"
val qty = 2.toString()
val xmlContent =
                 <basket>
                 <line>
                   <product qty={ qty }>{ productCode }</product>
                 </line>
               </basket>
println(xmlContent)
```

请注意，在XML元素中使用的所有变量都必须是字符串。另外请注意，在Scala中，字面量2是一个对象，因此可对其调用方法`toString()`。上述程序生成的输出如下：

```
<basket>
  <line>
```

```
      <product qty="2">PC Monitor</product>
    </line>
  </basket>
```

通过创建返回XML元素的函数，可轻松地将集合添加到XML输出中。返回XML元素的函数应返回一个`xml.Elem`实例：

```
def createXMLProduct(productCode: String): xml.Elem = {
  <product qty="1">{ productCode }</product>
}

val productCodes = List[String]("Keyboard", "Mouse")
def lines =
  <basket>
    <products> {
      productCodes.map(x => createXMLProduct(x))
    }</products>
  </basket>

println(lines.toString())
```

这里没有手动遍历集合，而是调用了方法`map`。对于集合中的每个元素，方法`map`都使用传递给它的lambda函数来返回新内容，从而对列表进行转换。在这里，它将包含`String`的列表`productCodes`转换为一个包含XML元素的新列表。函数`map`很好地展示了下一节将更详细地讨论的函数式编程。上述脚本的输出如下：

```
<basket>
  <products>
    <product qty="1">Computer Keyboard</product>
    <product qty="1">Mouse</product></lines>
  </products>
</basket>
```



注意，在实际编程中，不应以硬编码的方式将数量（属性`qty`）指定为`1`。

5.7　Scala 的函数式编程功能

前面说过，函数式编程的思维模式与命令式编程不同。这里介绍几个与函数式编程相关的主题：

- 使用函数遍历集合；
- 映射–过滤–归约设计模式；
- 柯里化。

5.7.1 使用函数遍历集合

在函数式编程中，很少使用for或while循环来遍历数组和集合，并在循环体中处理每个元素。相反，将对数组或集合实例调用一个在内部对其进行遍历的方法，这个方法将一个lambda函数作为参数，并对每个元素调用这个函数：

```
var a = List[Int](5, 10, 15, 20, 25)
a.foreach((x: Int) => println("%03d".format(x)))
```

这将打印005、010、015、020和025，其中使用了Java类java.lang.String的方法format来确保打印出来的整数包含三位（不够时在前面添加零）。

5.7.2 映射–过滤–归约设计模式

与函数式编程相关的一种著名的设计模式是映射–过滤–归约。下面分别来介绍这些模式：

- 映射；
- 过滤；
- 归约。

1. 映射——对数据进行变换

需要对数组或集合的每个元素都进行变换时，可使用方法map：

```
var a = List[Int](1, 2, 3)
var b = a.map((x: Int) => 2 * x)
println(b)
```

这将打印List(2, 4, 6)。

方法map将一个这样的lambda函数作为参数，即包含一个类型与数组或集合元素相同的输入参数。在这里，我们创建了一个lambda函数，它接受类型为Int的参数x，并返回2 * x。对于每个元素，函数map都将调用传入的函数，从而创建一个新的列表。

2. 过滤——过滤集合或数组中的元素

数组和集合类都实现了方法filter，使用它可将集合或数组中的元素剔除，这是通过传递一个函数实现的：这个函数返回一个布尔值，指出是否要将传递给它的元素保留在过滤后的列表或数组中：

```
var a = List[Int](100, 150, 200, 300)
var b = a.filter((x: Int) => x > 150)
println(b)
```

打印的结果为List(200, 300)。由于100和150都不大于150，因此对于这些元素，传递给

filter的函数将返回false，导致它们都不会添加到结果中。

3. 归约——执行计算

传递给方法reduce的lambda函数接受两个参数：初值和当前元素。返回的值将作为下次调用的初值。下面的示例将元素的值累加：

```
var a = List[Int](10, 20, 30, 40, 50)
var b = a.reduce((x: Int, y: Int) => x + y)
println(b)
```

这将打印结果150。再来看一个示例，它使用了Scala运算符max：

```
var a = List[Int](100, 2, 30, 60, 555)
var b = a.reduce((x: Int, y: Int) => x max y)
println(b)
```

这将打印555。运算符max返回两个值中较大的那个。正如你可能预期的，Scala还提供了运算符min。



与本章介绍的其他所有运算符一样，在Int类中，运算符min和max也是以方法的方式实现的。

5.7.3　柯里化

在Scala中，可向方法或函数提供多个参数列表：

```
class CurryingTest {
  def curryingMethod(a: Int, b: Int)(c: Int): Int = {
    a * b * c
  }
}
```

这被称为**柯里化**（currying）。在上述方法定义中，有两组输入参数，其中一组包含参数a和b，而另一组只包含参数c。调用这个方法时，要指定所有的参数，必须像下面这样做：

```
var c = new CurryingTest()
var result = c.curryingMethod(2, 3)(4)
println(result)
```

与预期的一样，这将返回24。如果只能在程序中这样调用这个方法，柯里化将毫无意义。在需要将函数传递给方法或函数时（这在函数式编程中很常见），柯里化很有用。下面是一个这样的示例：

```
def doCurrying(x: Int, fun: Int => Int): Int = {
  fun(x)
}
```

```
var result = doCurrying(30, c.curryingMethod(10, 20))
println(result)
```

这将打印6000。下面来粗略地解释一下这些代码。

(1) 方法doCurrying接受两个参数。

- x：一个Int实例。
- fun：接受一个Int参数并返回一个Int实例的函数。

(2) 函数doCurrying调用函数fun，将x作为输入参数传递给它，并返回函数fun的值。

(3) 调用函数doCurrying时，Scala创建一个临时的匿名（意味着没有名称）函数，这个函数使用第一组参数（a=10和b=20）调用传入的方法curryingMethod。创建的函数还需要一组参数（在这里，这组参数值只包含参数c），这样它才能执行curryingMethod。因此，生成的函数需要一个Int输入参数。

(4) 由于生成的匿名临时函数的签名与定义Int => Int兼容（接受一个Int输入参数并返回一个Int值），Scala编译器将这个生成的函数作为doCurrying的输入参数。

(5) 方法doCurrying执行传递给它的函数（这里是生成的匿名函数），并将x作为输入参数传递给它。至此，两组参数都有了，生成的匿名函数可以使用全部三个参数调用方法c.curryingMethod了。编译器知道这将返回一个Int，与doCurrying的返回类型相同。

5.8 小测验

(1) Scala是纯粹的OOP和函数式编程语言吗？

a) 是的，Scala是纯粹的OOP语言，也是纯粹的函数式编程语言。
b) 否，Scala是纯粹的OOP语言，但不是纯粹的函数式编程语言。
c) 否，Scala是纯粹的函数式编程语言，但不是纯粹的OOP语言。
d) 否，Scala既不是纯粹的OOP语言，也不是纯粹的函数式编程语言。

(2) 下面的代码能够通过编译吗？如果不能，请说明原因。

```
class A { def method1 = {} }
trait B { def method2 }
trait C { def method3 }
abstract class D extends A with B with C { }
```

a) 是的，这些代码能够通过编译并正确地运行。
b) 否，要实现特质，必须使用关键字implements。
c) 否，D类没有实现A类和/或特质B和C的方法。
d) 答案b和c都对。

(3) 下面的类定义有什么问题？

```
public class A
```

a) 这个类能够通过编译，没有任何问题。
b) 这个类不能通过编译，因为没有指定类体（`= { }`）。
c) 这个类不能通过编译，因为Scala中没有访问限定符`public`。
d) 答案b和c都对。

(4) 在函数式编程中，哪种设计模式最适合用来计算只包含`Int`值的数组的元素和？

a) 映射设计模式。
b) 过滤设计模式。
c) 归约设计模式。
d) 以上答案都不对。

(5) 在多线程程序中，对单例对象的可修改变量进行修改是安全的吗？

a) 是的，这种说法完全正确，因为Scala会确保每个线程都有自己的单例数据副本。
b) 不对，对于多个线程使用的单例对象，对其可修改的变量进行修改是不安全的。

5.9　小结

本章深入地介绍了Scala，它既提供了强大的函数式编程支持，又是一种纯粹的OOP语言。我们首先介绍了如何安装Scala以及如何使用命令`scala`——Scala的交互式REPL shell。通过使用这个功能强大的程序，你尝试了本章所有的代码片段。我们阐述了命令式编程和函数式编程的不同之处，探索了Scala语言众多的OOP功能，并发现很多Scala语句的功能比相应的Java语句更强大。我们还发现，Scala和Java的访问限定符很像，但又不完全相同。你尝试使用了Scala标准库中的一些集合类来进行泛型编程；最后，本章详细地探讨了函数式编程。

掌握这些理论知识后，该使用Scala来开发一个小型项目了，为此，我们将使用编译器`scala`以及Scala构建工具sbt。

第 6 章

Scala编程

6

本章将使用流行的工具包Akka创建一个小型的Scala项目。Akka旨在让你更轻松地创建可伸缩的JVM应用程序，它支持Java和Scala；由于是Scala之父开发的，因此非常适合用于创建Scala项目。

本章将创建一个简单的程序，它从一个固定的引言列表中随机地选择并显示一条引言。Akka基于本章将介绍的Actor模型。我们将使用Scala IDE来编写这个项目，Scala IDE存在独立的软件包和Eclipse IDE插件，我们将使用后者。我们将使用Scala构建工具（Scala Build Tool）来构建这个项目。本章介绍如下主题：

- Scala IDE for Eclipse插件；
- Scala构建工具（SBT）；
- SBT Eclipse for SBT插件；
- 编译器`scalac`；
- Akka工具包；
- 使用ScalaTest进行单元测试；
- 编写可执行的主应用程序。

6.1 Scala IDE for Eclipse 插件

默认情况下，Eclipse IDE不支持Scala。要让它支持Scala，需要安装Scala IDE插件。安装的Scala IDE决定了将在Eclipse IDE中使用哪个版本的Scala，但有时候，Scala IDE小组需要过段时间才能支持最新的Scala版本。安装Scala IDE后，就可在Eclipse中切换到新的<Scala>透视图。

6.1.1 安装 Scala IDE for Eclipse

可通过Eclipse Marketplace来安装Scala IDE，但这样安装的Scala IDE基于的Scala版本通常不是最新的。鉴于安装的Scala IDE版本决定了将支持哪个Scala版本，建议你安装Scala IDE小组管

理的仓库网站中最新、最稳定的版本。

通过手动添加Scala IDE的仓库，可让Eclipse下载并安装最新的稳定版本。为此，需要知道你使用的是哪个版本的Eclipse，这可通过在Eclipse IDE中选择菜单Help>About来获悉。通常，只需关注主版本号和次版本号，例如，在我的系统中，安装的是Eclipse 4.6版。

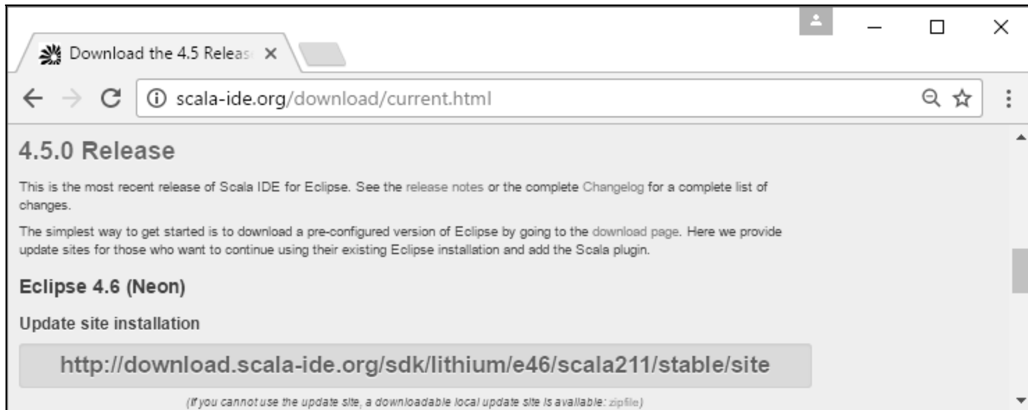


最新版的Scala IDE可能不支持最新的Scala版本，但Scala IDE小组通过其仓库发布更新的速度可能比Eclipse Marketplace快。

请访问Scala IDE网站（http://scala-ide.org）。

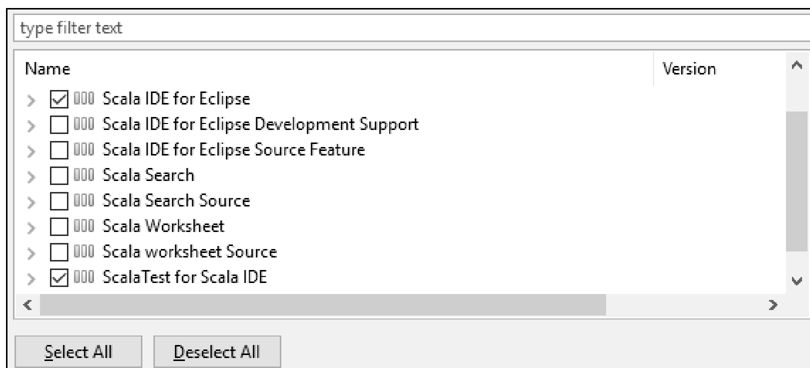
按如下说明在Eclipse IDE中安装Scala IDE插件。

- 在Scala IDE主页中，找到链接update sites。编写本书期间，这个链接位于醒目的按钮Download IDE旁边。
- 在支持的Eclipse版本列表中，找到你安装的Eclipse版本，并将Update site installation下方的链接复制到剪贴板中：



如果没有关闭Eclipse IDE，就切换到它；否则启动Eclipse IDE。然后，按如下说明将这个仓库添加到Eclipse中。

- 选择菜单Help>Install New Software...。
- 在出现的Install对话框中，Work with旁边的下拉列表包含已知仓库网站的清单，请单击这个下拉列表旁边的Add...按钮，以便添加Scala IDE仓库。
- 在出现的Add Repository对话框中，在文本框Name中输入Scala IDE，并在文本框Location中粘贴前面复制到剪贴板中的URL，再单击OK按钮保存这个仓库。
- 当前选择了仓库Scala IDE。在这个仓库的可用组件列表中，找到并选择Scala IDE for Eclipse和ScalaTest for Scala IDE：

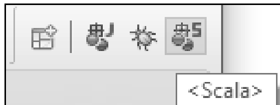


- 单击按钮Finish并按提示操作。

6.1.2 切换到 Scala IDE 透视图

Eclipse支持很多不同的编程语言，而安装Scala IDE后，它也将支持Scala。通过提供不同的透视图，Eclipse可针对特定的编程语言或环境优化用户界面。Scala IDE在Eclipse中添加了透视图Scala IDE，要切换到这个透视图，可按下面这样做。

- 在屏幕右上方，有一些表示最近使用的透视图的按钮。将鼠标指向这些按钮，有一个按钮会显示工具提示“Scala”。单击这个按钮：



- 如果这里没有表示Scala透视图的按钮，就单击显示工具提示“Open Perspective”的按钮，并从列表中选择Scala，再单击Open按钮。这将在工具栏中添加表示这种透视图的按钮。

选择透视图Scala后，将为Scala编程优化Esclipe IDE的用户界面。



可随时切换到不同的透视图。Eclipse IDE非常适合用于创建使用不同编程语言的项目。

6.2 SBT

要构建Scala项目，可使用大部分基于JVM的构建工具，包括Apache Maven和Gradle（第4章构建Java项目时使用的构建工具），但Scala提供了自己的构建工具——SBT（Scala Build Tool）。

Scala IDE默认并不支持SBT。稍后你将看到，这不是问题，因为可以反过来做：让SBT支持

Eclipse。我们将使用SBT新建一个项目，并安装一个能够创建并更新Eclipse项目的SBT插件。本节介绍如下与SBT相关的主题：

- 安装SBT；
- 新建基于SBT的项目；
- 添加在SBT中添加与Eclipse相关命令的SBT插件。

6.2.1　安装 SBT

要安装SBT，请访问http://www.scala-sbt.org，并下载用于你使用的操作系统的最新版本。

对于Windows操作系统，提供了负责安装并设置环境变量`Path`的MSI安装程序；对于其他操作系统，必须下载并解压缩一个归档文件（ZIP或TGZ），再将其位置添加到环境变量`Path`中。

与Gradle一样，SBT命令也可以从命令行执行，方法是将命令指定为参数。SBT的独特之处在于，它还提供了一个交互式shell。在交互式模式下运行SBT时，可以交互的方式执行SBT命令；另外，这个shell还支持自动补全——你只需按Tab键即可。

要检查安装情况，可在命令提示符（Windows）或终端（macOS和Linux）中输入`sbt`并按回车，这将以交互模式启动SBT。

6.2.2　创建基于 SBT 的 Eclipse IDE 项目

前面说过，Scala IDE for Eclipse插件存在的一个严重问题是，它当前不支持SBT。通过在SBT中安装一个插件，可从SBT生成Eclipse项目文件。要新建可在安装了Scala IDE插件的Eclipse IDE中打开的基于SBT的项目，可采取如下工作流程。

(1) 使用项目模板新建一个基于SBT的项目。
(2) 在SBT中添加插件sbteclipse。
(3) 在SBT中，使用插件sbteclipse生成一个新的Eclipse IDE/Scala IDE项目。

1. 新建SBT项目

要新建项目，最简单的方式是让SBT生成一个Hello World项目（其中包含一个空的构建文件）。必须在Eclipse IDE的workspace目录（Eclipse IDE用来存储项目的目录）中执行创建新项目的命令。如果你不确定这个目录的位置，可启动Eclipse并尝试创建一个Java项目，这样将显示目录workspace的路径。要新建一个基于SBT的项目，请按如下步骤操作。

- 启动命令提示符（Windows）或终端（macOS和Linux），并切换到目录workspace，再输入如下命令并按回车：

```
sbt new sbt/scala-seed.g8
```

- 首次执行这个命令时，SBT会下载一些依赖项。过段时间后，它将要求你提供项目名。请输入`Akka Quotes`并按回车。这将在目录akka-quotes中新建一个项目。
- 切换到目录akka-quotes，再输入`sbt`并按回车，以启动SBT的交互式shell。
- 输入下面的命令并按回车来尝试运行生成的项目：

```
run
```

同样，首次运行项目时，SBT将下载一些依赖项。下载完毕后，你将在控制台中看到一条“hello”消息：

```
Command Prompt - sbt

C:\Users\Vincent\workspace\akka-quotes>sbt
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
[info] Loading project definition from C:\Users\Vincent\workspace\akka-quotes\project
[info] Updating {file:/C:/Users/Vincent/workspace/akka-quotes/project/}akka-quotes-build...
[info] Resolving org.scala-sbt.ivy#ivy;2.3.0-sbt-2cf13e211b2cb31f0d3b317289dca70[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Compiling 1 Scala source to C:\Users\Vincent\workspace\akka-quotes\project\target\scala-2.10\sbt-0.13\classes...
[info] Set current project to Hello (in build file:/C:/Users/Vincent/workspace/akka-quotes/)
> run
[info] Updating {file:/C:/Users/Vincent/workspace/akka-quotes/}root...
[info] Resolving jline#jline;2.14.1 ...
[info] Done updating.
[info] Compiling 1 Scala source to C:\Users\Vincent\workspace\akka-quotes\target\scala-2.12\classes...
[info] Running example.Hello
hello
[success] Total time: 20 s, completed Feb 8, 2017 10:21:27 PM
>
```

输入命令`exit`并按回车，以退出这个交互式shell。

用来生成这个项目的模板基于最新且稳定的Scala版本，因此必须检查安装的Scala IDE版本是否支持这个版本。为此，在Eclipse IDE中选择菜单Window>Preferences。

在左边的列表中，找到并展开条目Scala，再选择其中的条目Installations。将安装的Scala IDE支持的Scala版本都记录下来；在我的系统中，支持的版本为Scala 2.11.8和Scala 2.10.6。在文本编辑器中，打开目录akka-quotes中生成的构建文件built.sbt。在我的系统中，这个文件类似于下面这样：

```
import Dependencies._

lazy val root = (project in file(".")).
  settings(
    inThisBuild(List(
      organization := "com.example",
      scalaVersion := "2.12.1",
      version := "0.1.0-SNAPSHOT"
    )),
    name := "Hello",
    libraryDependencies += scalaTest % Test
  )
```

如果其中的变量`scalaVersion`的值包含在安装的Scala IDE支持的Scala版本中，就万事大

6

吉。否则，你就必须将变量scalaVersion的值修改为相应的版本。对我来说，必须将这个变量修改成下面这样：

```
scalaVersion := "2.11.8",
```

如果必须修改这个变量的值，请在修改后从命令行运行如下命令，以清理并重新编译项目：

```
sbt clean run
```

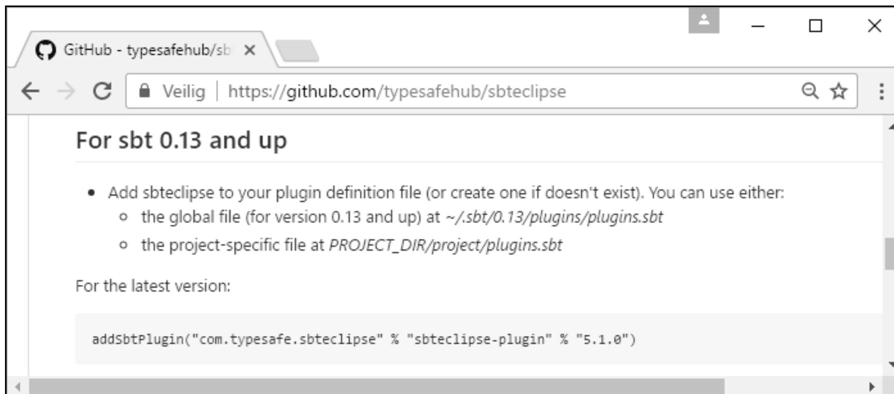
SBT将使用修改后的Scala版本重新编译项目。鉴于这个模板是基于最简单项目的，这通常可行——不会出现任何问题。



使用本章介绍的工具链时，如果出现严重的版本冲突问题，请尝试下载本书使用的版本，等你有了足够多的经验后，再切换到最新的版本。

2. 加载插件SBTEclipse

要获悉你必须安装该插件的哪个版本，请访问该插件的项目页面http://github.com/typesafehub/sbteclipse ）：



这个GitHub项目页面提供了如何在项目中添加这个插件的说明。你应查找以addSbtPlugin打头的行；我访问这个项目页面时，这行的内容如下：

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin"
             % "5.1.0")
```

在项目akka-quotes的子目录project中，新建一个名为plugins.sbt的文件，并将前述一行内容复制并粘贴到这个文件中，让SBT知道这个项目需要插件SBTEclipse。



务必将这个文件存储在子目录project中，否则SBT将找不到它。

3. 使用SBTEclipse生成新的Eclipse IDE项目

切换到目录akka-quotes，输入`sbt`并按回车以再次启动SBT交互式shell。

SBT将下载并激活插件SBTEclipse。从现在开始，在这个项目目录中启动SBT后，将能够创建或更新Eclipse IDE/Scala IDE项目。执行插件`sbteclipse`添加的如下命令：

```
eclipse
```

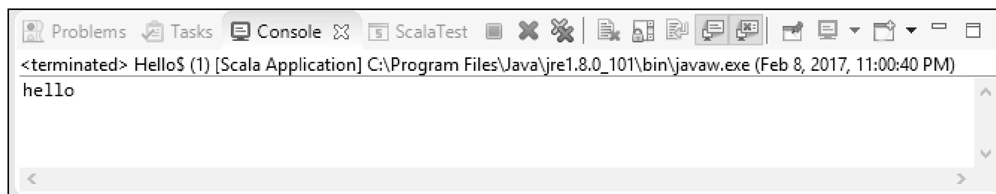
插件`sbteclipse`将在这个SBT项目的目录中生成项目文件，而安装了Scala IDE插件的Eclipse IDE能够导入的这些文件。

4. 在Eclipse IDE中导入生成的项目

返回到Eclipse IDE（如果它没有运行，就启动它）。为导入SBTEclipse生成的项目，请执行如下步骤。

(1) 在屏幕左边的Package Explorer的空白区域右击鼠标，并选择Import...。
(2) 将出现Import对话框，让你选择一个导入向导。请选择General>Existing Projects into Workspace，并单击按钮Next。
(3) 单击Select root directory旁边的按钮Browse...，选择目录workspace下的子目录akka-quotes，并单击OK按钮。
(4) 导入向导对话框的项目列表中将包含项目akka-quotes。单击按钮Finish关闭这个对话框。

这个项目将出现在Package Explorer中。在Package Explorer中，选择文件src/main/Scala>example>Hello.scala，再按Ctrl + F11运行这个文件。你也可单击工具栏中的Run按钮或选择菜单Run>Run来运行这个文件。如果一切顺利，你将在Console选项卡中看到问候“hello”：



6.2.3 Scala 编译器（scalac）

构建项目时，SBT将替我们调用Scala编译器`scalac`，因此我们无需直接调用它。但你必须知道，SBT使用的是`scalac`，而不是前一章一直使用的命令`scala`。

最重要的差别在于，Scala编译器像Java编译器`javac`一样，要求将代码封装在类中。命令`scala`通过在幕后创建一个不可见的类来满足这种要求，但编译器`scalac`不会这样做，因此在

Scala IDE中编写由SBT构建的Scala代码时，必须定义类并将代码添加到其中。

为此，有两种办法：

- 创建包含方法`main()`的单例对象；
- 让单例对象扩展特质`App`。

1. 创建包含方法main()的单例对象

这与Java很像。鉴于Scala没有与Java访问修饰符`static`等价的东西，因此必须使用单例对象。方法`main()`必须将String数组（`Array[String]`）作为输入参数，且返回类型为`Unit`（类似于Java关键字`void`）：

```
object MainObject {
  def main(args: Array[String]): Unit = {
    println("Executable code here...")
  }
}
```

2. 创建扩展特质App的单例对象

可不给单例对象添加方法`main()`，而使用Scala提供的特质`App`。`App`特质的不同寻常之处在于，你不能重写其中的任何方法，也不能在实现了特质`App`的类中添加方法`main()`。相反，你只需在类中直接添加可执行的代码，就像它们是类的主构造函数的可执行代码一样：

```
object MainObject extends App {
  println("Executable code here...")
}
```

本章将采用这种方法。

6.3　创建 Akka 项目

Akka是一个模块化工具包，用于创建健壮的分布式应用程序。它使用本章后面将深入介绍的Actor模型，并大量地使用Scala的函数式编程功能。这个库很大，因此本章只能演示其中的很小一部分。

有关这方面的更详细的信息，请访问Akka网站（http://akka.io/）。

建议你在创建这个项目期间将Akka文档放在身边。有关Akka文档的更详细信息，请参阅http://akka.io/docs/。

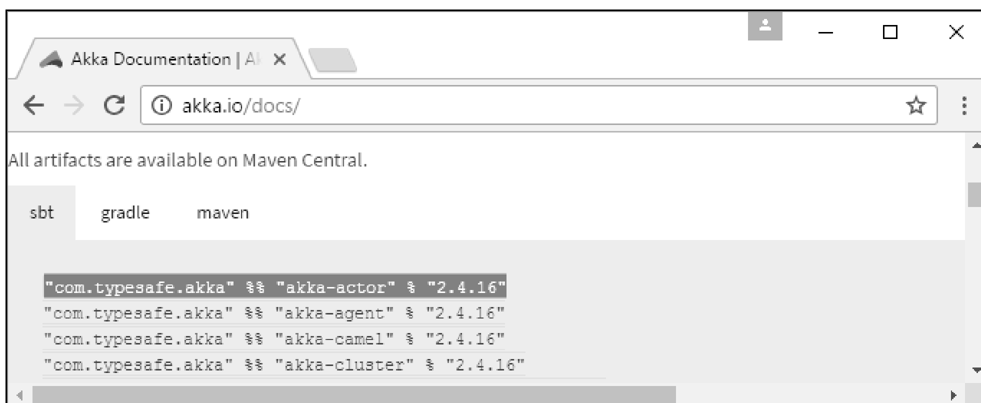
本节介绍如下主题：

- 在SBT构建文件中添加Akka依赖项；

- ❑ 更新Scala IDE项目；
- ❑ Akka概念；
- ❑ 创建Actor；
- ❑ 创建消息；
- ❑ 使用ScalaTest库对Actor进行单元测试；
- ❑ 编写可运行的应用程序。

6.3.1 在 SBT 构建文件中添加 Akka 依赖项

在Akka主文档网站，找到有关sbt的部分：



在列表中找到工件akka-actor，并将相应的行复制到剪贴板：

```
"com.typesafe.akka" %% "akka-actor" % "2.4.16"
```

在Eclipse IDE中，打开Package Explorer中的文件build.sbt，并对其做如下修改。

- ❑ 在包含libraryDependencies的那行末尾添加一个逗号。
- ❑ 添加一个空行，在其中输入libraryDependencies +=，再粘贴刚才从Akka文档中复制的内容，并在行尾添加一个逗号。

在前述工件列表中，找到akka-testkit，并重复刚才介绍的过程。确保最后一行末尾没有逗号，再在这行末尾添加% Test。

执行这些修改后，我的构建文件类似于下面这样：

```
import Dependencies._

lazy val root = (project in file(".")).
  settings(
```

```
inThisBuild(List(
  organization := "com.example",
  scalaVersion := "2.11.8",
  version := "0.1.0-SNAPSHOT"
  )),
  name := "Hello",
  libraryDependencies += scalaTest % Test,
  libraryDependencies += "com.typesafe.akka" %% "akka-actor"
                         % "2.4.16",
  libraryDependencies += "com.typesafe.akka" %% "akka-testkit"
                         % "2.4.16" % Test
)
```

通过在`libraryDependencies`条目末尾添加`% Test`，可让SBT确保相应的依赖只用于单元测试。运行主程序时，这些依赖项不会添加到classpath中，也不会随项目一起分发。按Ctrl + S保存所做的修改。

6.3.2　更新 Scala IDE 项目

更新文件build.sbt后，必须运行SBT，让它下载依赖项并将其添加到classpath中。在这里，还需使用插件sbteclipse更新Eclipse IDE项目，否则Eclipse IDE将看不到所做的修改。

在命令行中，切换到目录akka-quotes，再执行如下命令，让SBT下载新增的依赖并更新Eclipse项目：

```
sbt eclipse
```

SBT将下载新增的依赖项（以及它们的依赖项），还将相应地更新Eclipse IDE项目。过一会儿，你将看到如下消息：

```
[info] Successfully created Eclipse project files for project(s):
[info] Hello
```

在Eclipse IDE中，右击Project Explorer中的项目akka-quotes并选择Refresh。如果一切顺利，你将在Package Explorer的Referenced Libraries部分看到条目akka-actor。

6.3.3　Akka 概念

编写使用Akka工具包的Scala代码前，先来介绍一些Akka概念。前面说过，Akka使用了Actor模型。要编写Akka代码，你必须了解一些背景信息。本节介绍如下Akka概念：

- Actor；
- Actor引用（`ActorRef`）；
- 消息；
- 调度器。

1. Actor

在Actor模型中，不直接调用类的方法，而向Actor发送消息。在Akka中，Actor是扩展了特质`akka.actor.Actor`并包含消息处理程序的类实例。消息处理程序是一个方法，能够处理Actor支持的所有消息。Actor的消息处理程序可将收到的消息传递给其他Actor，也可创建一条或多条新消息并将其传递给其他Actor，还可创建新的Actor。

到目前为止，我们没看到Actor模型有任何特殊之处，也没有看到它相对于直接调用方法有任何优点。Akka的一个强大功能是，Actor不仅可运行在单个应用程序的单个进程中，还可运行在不同的线程和进程中（如不同的JVM实例中）。Actor甚至还可运行在网络上的不同计算机中。只要应用程序按规则行事，开发人员就无需纠缠于与并行/并发编程相关的经典问题，如加锁、避免竞态条件等，而这些问题常常难以重现和调试。

前面说过，Actor可创建自己的Actor。创建其他Actor的Actor是子Actor的父Actor，也被视为所有子Actor的监管Actor。如果子Actor失败，它将向监管Actor发送消息，而后者将负责处理问题。监管Actor可采取如下方式来处理错误：

- 请求子Actor接着往下执行任务，并保持其状态不变；
- 请求子Actor重新执行任务，并清除其状态；
- 永久性停止任务；
- 向上提交问题；此时任务将失败。

为创建本地Actor，最简单的方法是使用`ActorSystem`实例的工厂方法`actorOf`。为此，必须将`akka.actor.Props`类作为参数，它包含指向要创建的Actor所属类的引用。`Props`是一个向方法`actorOf`提供配置数据的类，如下例所示：

```
import akka.actor.{ ActorSystem, Actor, Props }

val system = ActorSystem("AkkaQuote")

class MyActor extends Actor {
  def receive: Actor.Receive = { Actor.emptyBehavior }
}

val myActorRef = system.actorOf(Props[MyActor], "My-Actor")
```

现在不用输入上述代码。

2. Actor引用（**`ActorRef`**）

通常情况下，Akka没有提供直接访问Actor实例的途径，而是在创建Actor时返回一个`ActorRef`实例，用于向相应的Actor发送消息。请注意，`ActorRef`实例没有暴露Actor的内部细节。`ActorRef`是一个不可修改的、线程安全的对象，可通过消息将其安全地传递给其他Actor。

6

Akka返回ActorRef实例而不是Actor类本身的实例的原因之一是，Actor可能运行在网络上的不同计算机中。通过使用ActorRef实例，可访问每个Actor，而不管它运行在应用程序所在的进程中，还是运行在远程服务器上。

Actor的消息处理程序能够访问变量self和sender，其中self包含指向当前Actor的ActorRef实例的引用，而sender包含指向发送消息的Actor的ActorRef实例的引用。



单元测试可使用特殊类TestActorRef来访问Actor的内部结构，这将在本章后面演示。

3. 消息

消息可以是任何包含必要数据的类的实例。强烈建议只在消息中包含不可修改的数据，因为如果Actor可随便修改消息的状态，就可能引发典型的多线程问题，这在前一节讨论过。相反，状态应在Actor中处理。Scala中的Case类非常适合用作消息，因为Actor的消息处理程序能够轻松地处理这种类的实例。

当前，Akka不能保证目标Actor一定能够收到发送的消息。只有Akka模块persistence（通过第2章简要讨论过的对象关系管理器库与数据库交互的Akka模块）能够确保目标Actor一定会收到消息。然而，可在接收端和/或发送端添加一个自定义层来处理传输错误。

发送给Actor的消息存储在一个队列中，这个队列被称为Actor的邮箱。Akka提供了多种内置的邮箱实现，Actor可根据性能要求选择不同的实现。另外，在必要的情况下，也可创建自定义的邮箱实现。Akka将确保Actor尽快处理其邮箱内的消息。

4. 调度器

调度器是一个线程池，Akka使用它来执行各种管理任务。它们确保发送的消息将放到目标Actor的邮箱中、在邮箱中等待的消息将被Actor处理、Actor请求的回调将被调用等。

Akka提供了默认的调度器实现，但你可以选择其他实现，还可自己实现调度器。

6.3.4　创建第一个 Akka Actor——QuotesHandlerActor

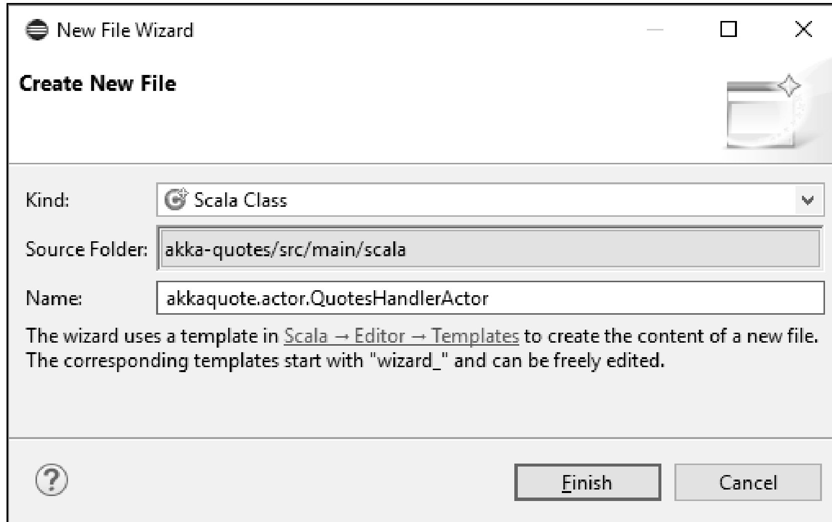
我们将编写一个简单的Akka程序，它在内存中存储一个名言列表。一个Actor负责管理这个列表（添加引言以及请求随机引言），而另一个Actor将请求一个随机的引言并将其打印到控制台。

首先，将目录main和test中既有的文件都删除。为此，在Eclipse的Package Explorer中，右击目录src/main/scala中的example包，并选择Delete。在被问及你是否确定要这样做时，单击OK按钮。

对目录src/test/scala中的example包做同样的处理。我们将重打锣鼓新开张。

接下来，新建一个名为`QuotesHandlerActor`的类，并将其放在`akkaquote.actor`包中。为此，右击Package Explorer中的目录src/main/scala，并选择New>Scala Class。

输入类名`akkaquote.actor.QuotesHandlerActor`：

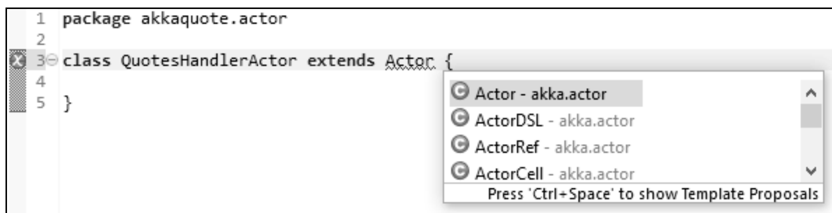


Scala IDE将创建指定的类和包，它生成的代码如下（删除了一些空行）：

```
package akkaquote.actor

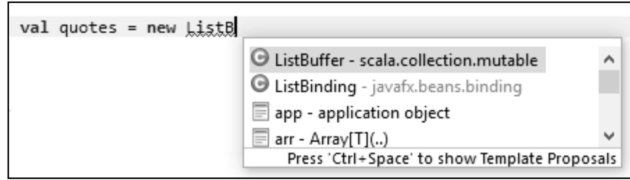
class QuotesHandlerActor {
}
```

将光标放在`QuotesHandlerActor`后面，添加一个空格并输入`extends Actor`，再按Ctrl+空格键。Scala IDE将显示一个列表，其中包含推荐的类名：



选择`akka.actor`包中的`Actor`类，并按回车。Scala IDE将自动添加相应的`import`语句。

这个类将用于存储引言，因此我们来实现一个可修改的列表。在这个类中，输入`val quotes = new ListB`并按Ctrl+空格键。将鼠标指向`scala.collections.mutable`包中的`ListBuffer`，并按回车：



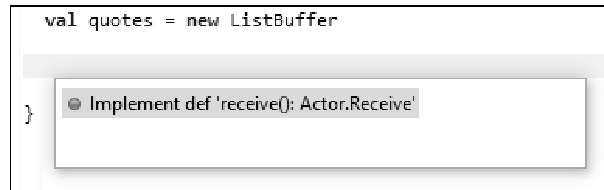
这行代码将变成`val quotes = new ListBuffer`。现在，整个类的代码应类似于下面这样：

```
package akkaquote.actor

import akka.actor.Actor
import scala.collection.mutable.ListBuffer

class QuotesHandlerActor extends Actor {
  val quotes = new ListBuffer
}
```

现在添加两个空行，并按Ctrl + 1（在macOS上为cmd + 1），Scala IDE将显示一个可实现的方法列表。选择方法`receive`并按回车：



Scala IDE将为你编写如下方法签名：

```
def receive: Actor.Receive = {
  ???
}
```

这是负责对收到的消息进行处理的方法。当前，还没有可处理的消息，但Akka要求你必须返回一个有效的对象。将`???`替换为`Actor.emptyBehavior`，这告诉Akka，这个方法没有任何行为，这是有意为之的。

要处理消息，得先编写它们。下面就来编写。

6.3.5　创建消息

前面说过，消息可以是任何实例，它们所属的类无需扩展任何特质或基类，但强烈建议你使用`Case`类。这是因为Actor只有一个`receive()`方法，所有的消息都将发送给它。稍后你将看到，使用`Case`类的模式匹配功能来处理消息非常方便。

请创建一个新类，将其命名为Messages，并放在akkaquote.message包中。为节省空间，我们将在这个文件中定义所有的公有消息类。不同于Java，Scala允许在同一个源代码文件中包含多个公有类。在这个文件中，将生成的整个Messages类删除。我们首先来编写import语句以及定义引言的类：

```
package akkaquote.message

import akka.actor.ActorRef

class Quote(val quote: String, val author: String)
```

Quote类定义了由文本和作者组成的引言，这两个变量都是不可修改的。接下来，新增一行并添加一些Case类，它们定义了这个示例中的Actor将使用的消息。为此，在这个源代码文件末尾添加如下代码：

```
case class AddQuote(quote: Quote)
case class RequestQuote(originalSender: ActorRef)
case object PrintRandomQuote

case object QuoteAdded
case class QuoteRequested(quote: Quote, originalSender: ActorRef)
case object QuotePrinted
```


前两个类（AddQuote和RequestQuote）是可发送给前面定义的Actor QuotesHandlerActor的消息。第三个对象（PrintRandomQuote）是可发送给后面将创建的ActorQuotePrinterActor的消息。其他三个对象是可作为应答发回的消息。对于前述代码，需要注意的一些细节如下。

- 这些代码非常紧凑。在Scala中，主构造函数是在类定义中定义的，因此无需编写定义字段和存储构造函数参数的代码，这些工作由Scala负责。
- 这些类和对象都无需对字段做额外处理，因为都不需要类体。
- 不需要参数的消息被定义为单例对象。并非必须这样做，但创建这些消息的多个实例只会浪费内存。
- 单例对象也可以是Case类。
- 消息RequestQuote和QuoteRequested的构造函数都将一个ActorRef实例（Actor引用）作为参数，其中的原因将在后面阐述。

还有一件事情要做——告诉QuotesHandlerActor只存储Quote实例。我们将使用泛型来完成这项工作。打开QuotesHandlerActor类并找到如下代码行：

```
val quotes = new ListBuffer
```

将其改成下面这样：

```
val quotes = new ListBuffer[Quote]()
```

请使用组合键Ctrl＋空格来输入类名Quotes，这样将自动添加所需的语句import akkaquote.message。

6.3.6　编写基于 ScalaTest 的单元测试

为演示单元测试库ScalaTest的工作原理，我们将编写一个对Actor进行测试的单元测试。为添加ScalaTest单元测试用例，请右击src/test/scala并选择New>Scala Class，再将这个类命名为QuotesHandlerActorTests，然后删除生成的代码，并开始编写必要的import语句以及类定义本身：

```
import akka.actor.{ActorSystem, Props}
import akka.testkit.{TestKit, ImplicitSender, TestActorRef}
import org.scalatest.{Matchers, FlatSpecLike, BeforeAndAfterAll}
import akkaquote.actor.QuotesHandlerActor
import akkaquote.message.{AddQuote, Quote, QuoteAdded}

class QuotesHandlerActorTests()
extends TestKit(ActorSystem("Tests"))
with ImplicitSender with Matchers
with FlatSpecLike with BeforeAndAfterAll {

}
```

这个类扩展了Akka模块TestKit中的TestKit类，后者提供了大量让你能够更轻松地对Akka Actor进行测试的方法。通过实现特质FlatSpecLike和Matchers，可使用ScalaTest的DSL以更自然的方式编写单元测试。实现特质ImplicitSender确保这个类将收到Actor以应答方式发送的消息。

所有测试都结束后，妥善地停止Akka系统至关重要，否则将导致内存泄露。在这个类中，添加如下方法：

```
override def afterAll(): Unit = {
  system.terminate()
}
```

之所以可以重写方法afterAll()，是因为这个类实现了特质BeforeAndAfterAll。字段system是从TestKit类那里继承而来的。

在我们的测试中，将测试我们能够通过向Actor QuotesHandlerActor发送AddQuote消息来添加新引言；为此，在这个类中添加如下代码。注意，我们使用了ScalaTest的**领域特定语言**（domain-specific language，DSL）。乍一看，这些代码可能有点怪异：

```
"An QuotesHandlerActor" should "add new quotes" in {
  val quoteHandlerActorRef = TestActorRef(Props[QuotesHandlerActor])
  val actorInstance = quoteHandlerActorRef.underlyingActor
                                .asInstanceOf[QuotesHandlerActor]
```

```
    actorInstance.quotes.size should be(0)

    val quote = new Quote("This is a test", "me")
    quoteHandlerActorRef ! AddQuote(quote)
    expectMsg(QuoteAdded)

    actorInstance.quotes.size should be(1)
    actorInstance.quotes(0).quote should be("This is a test")
    actorInstance.quotes(0).author should be("me")
  }
```

前述代码执行了很多任务，下面来详细说说。

- 第一行定义了一个测试，其中的第一个字符串（`A QuotesHandlerActor`）指出了要测试的类或对象，该字符串后面是关键字`should`，它告诉ScalaTest这是一个测试。接下来的字符串指出了测试的是什么，这里为“收到消息`AddQuote`时添加一个新引言”。在关键字`in`的后面是一个代码块，其中包含测试的代码。
- 我们创建了一个`TestActorRef`对象，它指向`QuotesHandlerActor`。`TestActorRef`是一个可用于单元测试的`ActorRef`实例，其功能之一是让你能够直接访问它包装的Actor实例`QuotesHandlerActor`。
- 我们通过这个`TestActorRef`对象获取指向`QuotesHandlerActor`实例的引用。有了指向Actor对象的引用后，就可以测试其内部功能了。
- 通过使用Actor引用`actorInstance`，我们检查其集合成员quotes的长度是否为零。如果不是，这个测试将失败。
- 我们向`TestActorRef`实例发送消息`AddQuote`，该实例将把消息转发给实际的Actor。我们之所以这样做，是因为只有`ActorRef`类实现了运算符`!`。变量`actorInstance`指向一个Actor类实例，而`TestActorRef`是`ActorRef`类的一个实例。我们创建一条新引言，并将其作为消息的参数。
- 由于我们实现了特质`ImplicitSender`，因此这个类将收到消息`AddQuote`发回的应答。方法`expectMsg`检查是否收到了指定的消息，其默认超时时间为3秒，就这里而言，这足够了。
- 接下来，我们检查`actorInstance`中的列表`quotes`是否包含一个元素，并检查属性`quote`和`author`是否与我们发送给消息的引言匹配。



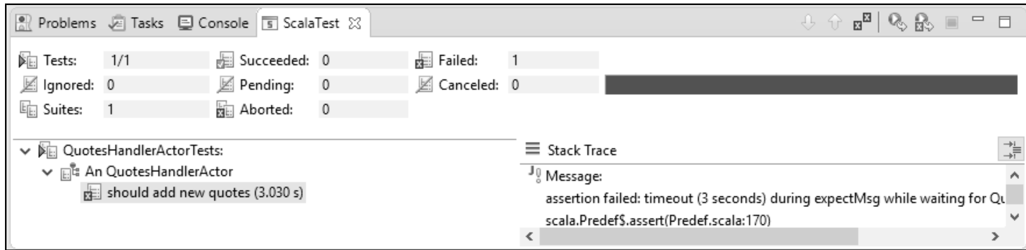
在有些Scala IDE版本中，存在一个与方法`expectMsg`相关的bug，即在这个方法可用的情况下显示错误消息`not found: value expectMsg`。如果你遇到这种错误，可置之不理。这样Scala IDE也将正确地编译并运行代码。

说得差不多了，下面来运行这个单元测试：在Package Explorer中右击`QuotesHandler-ActorTests`类并选择Run As>ScalaTest-File。测试以失败告终，这没有什么可奇怪的。控制台包

含的信息类似于下面这样：

```
Run starting. Expected test count is: 1
QuotesHandlerActorTests:
An QuotesHandlerActor
- should add new quotes *** FAILED ***
 java.lang.AssertionError: assertion failed: timeout (3 seconds) during
expectMsg while waiting for QuoteAdded
```

请切换到选项卡ScalaTest，以查看摘要：



我们还没有处理消息`AddQuote`，因此单元测试脚本没有在合理的时间内收到应答消息`QuoteAdded`。下面来改变这种现状。

6.3.7　实现消息处理程序

在Package Explorer中，打开`akkaquote.actor`包中的文件QuotesHandlerActor.scala，并在其中添加如下必不可少的`import`语句：

```
package akkaquote.actor

import akka.actor.Actor
import scala.collection.mutable.ListBuffer
import scala.util.Random
import akkaquote.message.{ Quote, AddQuote, QuoteAdded, RequestQuote,
                           QuoteRequested }
```

将方法`receive`的当前实现替换为如下代码：

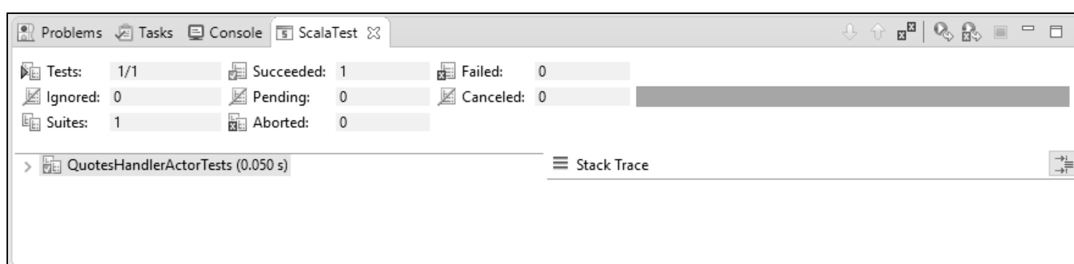
```
def receive = {
  case AddQuote(quote) => {
    quotes += quote
    sender ! QuoteAdded
  }

  case RequestQuote(originalSender) => {
    val index = Random.nextInt(quotes.size)
    sender ! QuoteRequested(quotes(index), originalSender)
  }
}
```

拜神奇的Case类和模式匹配所赐，我们能够轻松地检查收到的是哪种消息，并做相应的处理。收到消息AddQuote时，将传递的对象quote加入列表quotes，并发回应答消息QuoteAdded。

这个Actor还处理消息RequestQuote。收到这种消息时，它从列表中随机选择一条引言，并将其传递给应答消息QuoteRequested。该应答消息将发回给发送消息的Actor，其中包含ActorRef实例originalSender，让消息接收方能够将该应答消息发回给请求引言的Actor，后面将更详细地解释这一点。

请打开目录src/test/scala中的文件QuotesHandlerActorTests.scala，右击这个文件并选择Run As>ScalaTest-File，以再次运行测试。这次测试应该通过了。祝贺你成功了！



6.3.8 创建 QuotePrinterActor

下面来再来创建一个Actor，它向QuotesHandlerActor请求一条引言，并将返回的引言打印到控制台。这个Actor的工作原理如下：收到消息RequestQuote后，将其发送给QuotesHandlerActor。正如你在前面看到的，QuotesHandlerActor将发回应答消息QuoteRequested，其中包含一条随机的引言。收到这条消息后，我们要创建的Actor将把它打印到屏幕上。为节省篇幅，这里不对这个类做单元测试。



注意，在生产环境中，出于时间考虑而省却单元测试通常是非常糟糕的主意。

请新建一个类，并将其命名为akkaquote.actor.QuotePrinterActor。接下来，首先添加import语句和类体：

```
package akkaquote.actor

import akka.actor.{ Actor, ActorRef }
import akkaquote.message.{ PrintRandomQuote, RequestQuote,
                           QuoteRequested, QuotePrinted }

class QuotePrinterActor(val quoteManagerActorRef: ActorRef) extends
      Actor {
}
```

需要指出的是，这个类的主构造函数将一个ActorRef实例作为参数。通过使用ActorRef对象，可使用运算符!向这个Actor发送消息。

在这个类中，添加消息处理程序：

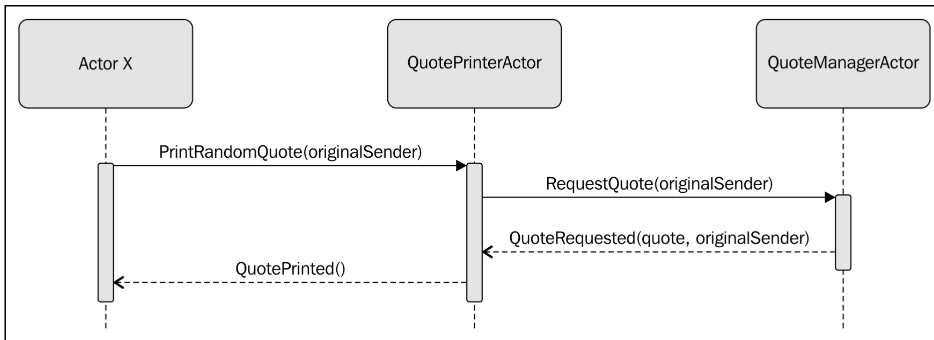
```
def receive: Actor.Receive = {
  case PrintRandomQuote => {
    val originalSender = sender
    quoteManagerActorRef ! RequestQuote(originalSender)
  }

  case QuoteRequested(quote, originalSender) => {
    System.out.println('"' + quote.quote + '"')
    System.out.println("-- " + quote.author)
    originalSender ! QuotePrinted
  }
}
```

收到PrintRandomQuote消息后，QuotePrinterActor向QuoteManagerActor发送RequestQuote消息，这是使用通过构造函数传入的Actor引用实例quoteManagerActorRef实现的。QuoteManagerActor将发回应答消息QuoteRequested，其中包含一条随机的引言。

收到QuoteRequested消息后，QuotePrinterActor将其打印到控制台。

在RequestQuote消息中添加了发送者，这看起来好像很复杂。下面来详细说说。收到PrintRandomQuote消息时，发送者是将该消息发送给QuotePrinterActor的Actor；此时不知道该Actor是谁，我们暂且称之为Actor X。下面的UML时序图说明了这个流程：



在RequestQuote消息中，以originalSender的方式添加了指向Actor X的引用。在发回给QuotePrinterActor的应答消息QuoteRequested中，QuoteManagerActor原封不动地传递了引用originalSender。将引言打印到控制台后，处理程序QuoteRequested可以向originalSender（Actor X）发送消息QuotePrint，让它知道向控制台打印了一条消息。如果使用Actor引用Sender而不是originalSender，QuotePrinted消息将发回给QuoteManagerActor，这并不是我们希望的。

6.3.9 主应用程序

下面来创建一个基于控制台的简单应用程序，它在集合中添加一些引言，并随机地打印一条。这个应用程序将通过向Actor发送消息来通信。在Actor中，消息处理程序不能是阻塞的，这意味着它不应在同一个线程中运行耗时的代码。我们的Actor符合这条规则，它们都直接处理消息并立即返回。

为了演示这两个Actor的工作原理，我们将在主程序中采取不同的做法：向一个Actor发送消息，并等到收到应答后再继续执行。这样做旨在确保添加所有引言后才请求打印引言。另外，我们还想在打印引言后再停止Akka系统和程序。

为此可使用ask模式。这种模式向Actor发送消息后立即返回，而不等待应答。它返回一个名为`Future`的对象的值；调度器可在独立的线程中运行`Future`对象，而不阻塞Akka应用程序。另外，我们也可等待`Future`对象中的代码运行完毕（暂停程序），直到收到应答或超时，这个示例将采取这种做法。

右击`akkaquote`包并选择New>Scala Object，以添加一个单例对象。将代码修改成下面这样：

```
package akkaquote

import akka.actor.{ ActorSystem, Props, ActorRef }
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.Await
import scala.concurrent.duration.DurationInt
import scala.language.postfixOps
import akkaquote.actor.{ QuotePrinterActor, QuotesHandlerActor }
import akkaquote.message.{ Quote, AddQuote, RequestQuote,
                           PrintRandomQuote }

object Main extends App {

}
```

这里包含的`import`语句很多。对Akka Actor来说，`akka.actor`包中的类至关重要。`akka.pattern.ask`包也必须导入，否则将无法识别方法`ask`或类似的运算符`?`。方法`ask`是一个开销很大的命令，Akka设计者希望程序员明白这一点。要等待`Future`对象执行完毕，`Await`类必不可少。为使用后缀表示法设置超时时间，必须导入`Timeout`、`DurationInt`和`postfixOps`类。

首先来初始化Akka和Actor，为此在这个对象中添加如下代码：

```
val system = ActorSystem("AkkaQuote")
val quoteActorRef = system.actorOf(Props[QuotesHandlerActor],
                                  "quotesActor")
val quotePrinterActorRef = system.actorOf(Props(new
                            QuotePrinterActor(quoteActorRef)),
                            "quotesPrinterActor")
```

接下来，添加初始化超时时间以及在QuotesHandlerActor中添加3条引言的代码：

```
implicit val timeout = Timeout(10 seconds)

val future1 = quoteActorRef ? AddQuote(new Quote("Hello world",
                                          "Various book authors"))
val future2 = quoteActorRef ? AddQuote(new Quote("To be or not to be",
                                           "W. Shakespeare"))
val future3 = quoteActorRef ? AddQuote(new Quote(
                     "In the middle of difficulty lies opportunity",
                     "A. Einstein"))

Await.result(future1, timeout.duration)
Await.result(future2, timeout.duration)
Await.result(future3, timeout.duration)
```

使用运算符?时（前面说过，必须导入akka.pattern.ask包，否则无法使用运算符?），将返回一个Future对象。对于每条消息，我们都等待应答或超时；如果超时，将引发异常。在这个小型应用程序中添加最后一段代码：

```
val future4 = quotePrinterActorRef ? PrintRandomQuote
Await.result(future4, timeout.duration)

system.terminate()
```

我们使用指向QuotePrinterActor的ActorRef实例向它发送PrintRandomQuote消息。同样，我们等待应答消息或超时。最后，调用方法terminate()来停止Akka系统。



有些Scala IDE版本存在bug，无法识别方法system.terminate()，进而显示错误消息“value terminate is not a member of akka.actor.ActorSystem”，但这些代码能够通过编译并正确地运行。

这个程序将优雅地退出并妥善地释放所有的资源。

如果你按Ctrl + F11运行这个应用程序，将看到它向控制台打印了一条引言：

Problems　Tasks　Console

<terminated> Main$ (2) [Scala Application] C:\Program Files\Java\jre1.8.0_101\bin\java

```
"Hello world"
-- Various book authors
```

6.4　小结

本章介绍了很多技术。我们安装了Eclipse IDE插件Scala IDE，以便在Eclipse IDE中编写Scala代码，并利用Eclipse的众多功能。为了构建项目，我们安装了SBT。此外还安装了SBT插件sbteclipse，因为Scala IDE不支持SBT。SBTEclipse用于创建和更新使用SBT构建文件的Scala IDE项目。

我们学习了Actor模型。在这种模型中，各种Actor相互发送消息，其中每个Actor都使用单个方法来处理所有的消息。不直接与Actor实例通信，而使用`ActorRef`实例。使用`ActorRef`实例时，代码不太关心Actor运行在本地还是远端。我们使用DSL编写了一个单元测试，以便对一个Actor进行测试。最后，我们编写了主程序，它使用ask模式和`Future`对象来等待应答。

下一章将详细介绍Clojure——用于JVM的Lisp语言实现。Clojure对Scala拥趸很有吸引力，因为它专注于函数式编程范式。

第 7 章 Clojure

Clojure与本书介绍的其他语言大不相同，其灵感主要来自20世纪50年代末推出的Lisp编程语言。Lisp在技术上与时俱进，现在依然没有过时。Common Lisp和Scheme无疑是当前最流行的两种Lisp方言。Clojure也是一种Lisp方言，但其设计深受这两种方言的影响。

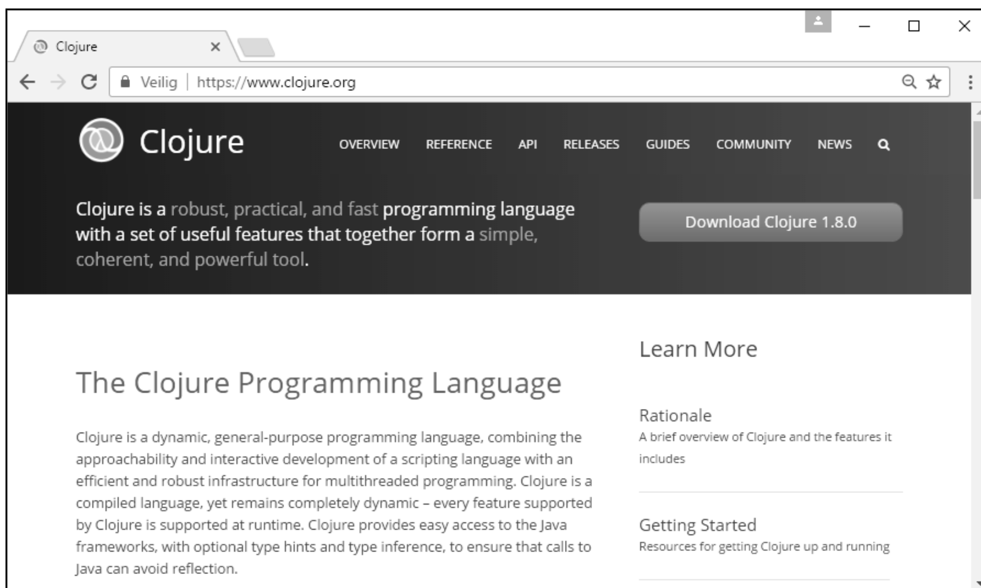
不同于Java和Scala，Clojure是一种动态语言：变量的类型不固定；编译时编译器不执行类型检查。将变量传递给函数时，如果它与函数中的代码不兼容，将在运行阶段引发异常。需要指出的是，不同于本书介绍的其他语言，Clojure并不是面向对象语言，但能够与Java和JVM互操作，因为它能够创建对象实例。另外，它还能够生成类，让其他与Java兼容的语言能够运行它生成的字节码。

本章介绍如下主题：

- 安装Clojure；
- Clojure的交互式shell——读取–评估–打印–循环（REPL）；
- Clojure基础知识；
- 使用类；
- Clojure代理系统（agent system）。

7.1 安装 Clojure

从官网（https://clojure.org/）下载最新的版本：



建议你在尝试本书的示例时参考Clojure文档，为此可访问在线官方文档或社区驱动的文档网站：

- http://clojure.org/reference/；
- http://clojure-doc.org。

编写本书期间，最新版为Clojure 1.8.0。将下载的归档文件解压缩，并将解压缩时指定的目标目录记录下来，供后面创建启动脚本时使用。

为验证是否成功地安装了Clojure，可尝试启动交互式shell。为此，在命令提示符（Windows）或终端（macOS和Linux）中切换到Clojure安装目录，并执行如下命令（请将版本号替换为你安装的版本）：

```
java -jar clojure-1.8.0.jar
```



就测试交互式shell而言，这个命令挺好，但运行Clojure代码时，不要使用它，因为使用了选项-jar时，无法设置自定义classpath，这在第2章详细解释过。后面将介绍一种更佳的运行交互式shell的方式。

如果一切顺利，控制台中将出现类似于下面的输出：

```
Clojure 1.8.0
user=>
```

请输入如下代码并按回车键退出shell：

```
(System/exit 0)
```

这些代码运行`java.lang.System`类的标准方法`exit`，从而妥善地停止当前JVM实例。

创建启动脚本

不同于众多其他的JVM语言，Clojure没有为常见的操作系统提供启动脚本。为了运行Clojure，最简单的方法是手动创建一个启动脚本，并将其放在环境变量Path包含的目录中。

在接下来将创建的启动脚本中，指定了固定的类路径。如果你使用Clojure创建的应用程序需要其他的类，建议为其创建一个自定义的启动脚本。

1. 在Windows中创建启动脚本

使用你喜欢的文本编辑器创建一个包含如下内容的文件：

```
java -cp c:\PATHTO\clojure\clojure-1.8.0.jar clojure.main
```

请将`c:\PATHTO\clojure`替换为Clojure安装路径，并将版本号替换为你安装的版本。

在命令窗口中执行命令`path`，以查看环境变量`Path`包含的目录。你也可将目录添加到环境变量`Path`中，详情请参阅第2章。将这个文件存储到环境变量`Path`包含的一个目录中，并将其命名为clojure.bat。

2. 在macOS和Linux中创建启动脚本

创建一个包含如下内容的文件：

```
java -cp /path/to/clojure/clojure-1.8.0.jar clojure.main
```

请将`/path/to/`替换为Clojure安装路径，同时将版本号替换为你安装的版本。将这个文件保存到环境变量`Path`包含的一个目录中，并将其命名为clojure.sh。为确保文件clojure.sh是可执行的，在终端中切换到它所在的目录，并执行如下命令：

```
chmod +x clojure.sh
```

7.2　Clojure 的交互式 shell（REPL）

现在可以使用这个启动脚本来运行Clojure的REPL交互式shell了。不同于Scala的REPL交互式shell `scala`，Clojure的REPL没有自己的自定义命令：正如你在前面看到的，要妥善地退出，必须调用`java.lang.System`类的方法`exit()`——输入代码`(System/exit 0)`并按回车：

```
C:\Java\Languages\clojure-1.8.0>java -cp clojure-1.8.0.jar clojure.main
Clojure 1.8.0
user=> (println "hello")
hello
nil
user=> (System/exit 0)

C:\Java\Languages\clojure-1.8.0>
```

Clojure没有自带独立的编译器命令，而在执行Clojure代码时在内存中生成并执行JVM字节码。要生成类文件并将其存储到文件系统中，以便供其他JVM语言使用，你必须调用普通的Clojure函数，这将在下一章演示。

7.3 Clojure 语言

鉴于Clojure与众多主流编程语言截然不同，我们将更详细地介绍Clojure基础知识，这包括：

- 语法；
- 表达式；
- 定义变量；
- 定义函数；
- 数据结构（数字、字符串和集合）；
- 数组迭代和循环；
- 条件。

7.3.1 语法

Lisp和Clojure都遵循**代码即数据、数据即代码**的原则。Lisp的这种性质被称为**同像性**（homoiconicity），这意味着语言的语法与使用该语言编写的程序的结构类似。Lisp的内置数据类型之一是列表，它用于编写代码。定义列表后，可在其中添加表达式。表达式包含一个函数引用及其参数。在运行阶段，到达列表结束位置后，将动态地执行列表。大致而言，整个程序都是由Clojure的内部数据结构表示的。Clojure包含一个名为阅读器（reader）的进程，它读取并执行每个列表项，再创建相应的数据结构并将其传递给编译器。下面来看一个简单的表达式示例：

```
(println "Hello" "from Clojure!")
```

请在Clojure的REPL shell中输入这些代码，REPL shell将在控制台中打印如下内容：

```
Hello from Clojure!
nil
```

字符(和)分别开始和结束列表。在这里，列表的内容是单个可被阅读器读取并执行的表达式。这个表达式包含对Clojure函数`println`的调用。函数`println`是一个**参数可变的**（variadic）函数，这意味着它将列表中的其他元素都作为输入参数。这有点像第3章简要讨论过的Java关键字`varargs`。在这里，有两个参数，这些参数将在评估后传递给函数`println`。函数`println`将传入的字符串打印到控制台，但什么都不返回（如果是在Java中，将使用关键字`void`来声明这个函数），因此调用这个函数的结果为`nil`——类似于Java中的`null`。不同于Java，Clojure中的函数调用都有结果。

你可能会问，与定义固定的语言语法，并让编译器使用传统分析程序进行分析相比，使用包含代码的数据结构（本章主要关注表达式）有何优点呢？答案是这样可编写直接操纵代码的特殊函数（为此只需修改数据结构即可），这些特殊函数被称为**宏**（macro）。宏通过操纵包含表达式的列表来修改或改进既有的代码。这提供了很多可能性。创建宏是一个很复杂的主题，本书不会介绍，而只关注Clojure提供的函数和宏。

7.3.2　表达式

表达式以函数调用打头，函数调用后面紧跟着参数。稍后你将看到，表达式可以嵌套。

不同于众多的主流语言，Clojure没有运算符的概念，而是提供了返回计算结果的函数。例如，它提供了函数+，用于将当前列表中指定的所有数值相加：

```
(+ 10 20 30)
```

这将返回`60`。

在Clojure中，每个表达式的结果都为单个值。表达式放在列表中；第一个列表项为函数，而其他列表项都是这个函数的参数。先计算参数的结果，再将其传递给函数，这使得可将参数指定为嵌套表达式：

```
(+ 10 ( * 3 5 ))
```

将按如下顺序计算上述表达式：

```
(+ 10 ( * 3 5 ))
(+ 10 ( 15 ))
(+ 10 15)
(25)
```

第二个参数是一个列表，其结果为15。注意，传递给函数+的第二个参数为15，因此整个表达式的结果为25。Clojure没有运算符优先级的概念，而只是按从左到右的顺序计算包含表达式的列表，因此执行计算时经常需要嵌套列表。

最重要的算术运算函数总结如下。

函　　数	描　　述	示　　例
+	将值相加	(+ 10 20) →+30
-	从左到右将值相减	(- 50 25) → 25
*	将值相乘	(* 10 20) → 200
/	将值相除（更详细的信息请参阅稍后将讨论的数值类型）	(/ 25 5) → 5
quot	求商	(quot 13 4) → 3
rem	求余	(rem -13 4) → -1
mod	求模	(mod -13 4) → 3
inc	将值加1	(inc 41) → 42
dec	将值减1	(dec 43) → 42
max	返回最大值	(max 100 20 30) → 100
min	返回最小值	(min 0 -1 30) → -1

7.3.3　定义变量

Clojure并非纯粹的函数式编程语言，它可创建在不同时间指向不同数据结构的变量，这意味着可能带来副作用。变量是使用函数def定义的：

```
(def var-name)
```

如果没有指定值，变量将是未绑定的。可在定义变量的同时指定值，从而让变量指向这个值。这将创建一个新的全局绑定：

```
(def var-name "This is a value")
```

当前，变量var-name指向字符串This is a value；要让它指向另一个值，可再次使用函数def：

```
(def var-name 100)
```

这不会修改之前对var-name的引用；只有以后的代码读取变量var-name时才能看到新指定的值100。

可在每个线程中绑定变量，这样每个线程都有自己的变量副本，但这不的本书的讨论范围之内。

7.3.4　定义函数

要创建函数，最简单的方式是使用函数defn：

```
(defn greet [name] (println (str "Greetings, " name "!")))
```

7

要调用前面定义的函数greet，可像通常那样做：

```
(greet "reader of this book")
```

这行代码向控制台打印如下内容：Greetings, reader of this book!。对于函数defn，有几点需要说明。

- 第一个参数为函数名。
- 第二个参数必须是一个vector对象，其中包含要定义的函数的输入参数。如果不需要参数，可使用空vector对象——[]。
- 第三个参数是函数要计算并返回的表达式。
- 返回最后一个表达式的结果。这里为nil，因为函数调用println的结果为nil。

要让函数greet返回true（而不是nil），应将其指定为最后一个列表项，且不能将其放在括号内，因为值true不是函数：

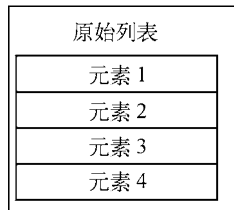
```
(defn greet [name] (println (str "Greetings, " name "!")) true)
```

7.3.5 数据结构

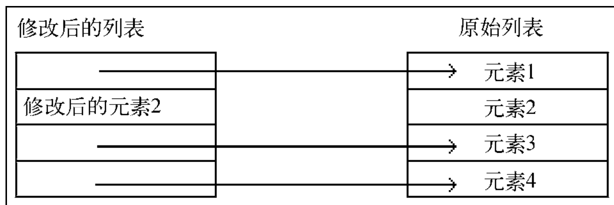
在介绍Scala的两章中说过，不可修改的数据是函数式编程的基石。因此，函数不修改数据，而是返回修改后的数据副本，以免修改既有的数据副本。在Scala中，不可修改的列表就是这样的：使用运算符+在不可修改的列表中添加新元素时，将返回一个新列表，而原来的列表保持不变。

乍一看，存储多个稍微不同的数据副本会消耗宝贵的内存，看起来是过于浪费，但实际上，Clojure通过使用复杂的数据结构，巧妙地使用引用字段，因此在修改后的数据副本中，只有发生了变化的数据需要占用额外的空间。

假设我们创建了一个简单的列表，它包含4个元素：



我们修改这个列表中的第二个元素，这导致Clojure创建一个新列表。在这个新列表中，除第二个元素外，其他元素都指向原始列表中的元素：



由于每个版本的数据都是不可修改的且永远不会发生变化，因此这些引用始终是有效的。



这被称为**永久性数据结构**，但不要将其与**对象关系管理器**（ORM）使用的持久化数据库对象混为一谈。

Clojure数据结构是良好的JVM公民，它们遵循第1章和第2章介绍的JVM约定，实现了方法`hashCode()`和`equals()`，因此可用于常规Java集合（如`HashMap`实例）中。另外，它们还使用JVM接口，从而向调用者隐藏了实现细节。Clojure的集合类实现了迭代器，因此可用于`for`循环中。为实现与Java生态系统的良好兼容性，Clojure开发小组花费了很大的精力，因此Clojure也能够与其他大多数流行的JVM语言很好地兼容。

1. 数值类型

出于性能和效率考虑，Clojure将JVM基本数据类型（而不是前几章介绍的包装类）作为默认的数值类型。在Clojure中，整数的默认类型为`long`。只要一个值可存储在`long`变量中，Clojure就会使用基本类型`long`，哪怕这个值使用`int`变量也能存储。对于浮点值，Clojure默认使用`double`变量来存储。后面你将看到，Clojure还支持标准Java类库中支持更大值和/或更高精度的类。



必须指出的是，Clojure也支持基本类型包装类。例如，使用`java.lang.Integer`类时，Clojure将其自动装箱为基本类型`int`。

当计算结果不为整数时，将返回一个`Ratio`对象，这是Clojure特有的功能。我们来看一个示例：

```
(/ 1 3)
```

在大多数语言中，整数1和3相除的结果为0（也是整数），但在Clojure中，结果如下，这可能有点出乎意外：

```
1/3
```

这是一个`Ratio`类实例。`Ratio`类是Clojure运行时库中定义的一个常规JVM类，它将分子和分母存储在不同的字段中。如果要获得双精度结果，而不涉及`Ratio`对象，必须至少将其中一个整数改为双精度值：

7

```
(/ 1 3.0)
```

这将返回你更熟悉的基本类型double值0.3333333333333333。

要计算商和模，可使用函数quot和mod：

```
(quot 42 10) (mod 42 10)
```

在这里，这两个函数的结果分别为4和2。

除了内置的基本数据类型long和double外，Clojure还提供了BigInt类型，其变量可存储的数字要比long变量大得多。Clojure还支持Java类库中java.math包中的Java类BigDecimal。通过在整数前面加上字母N，可让Clojure将其转换为BigInt实例：

```
(+ 100 1N)
```

这将返回101N。调用运算符函数时，只要有一个输入参数为BigInt实例，计算结果也将为BigInt实例。BigDecimal的工作原理与BigInt类似。要将一个值转换为BigDecimal实例，只需在它前面加上字母M：

```
(+ 555 0.4169M)
```

这将返回555.4169M——完全在意料之中。

Clojure文档指出，调用标准算术运算函数时，如果指定的整数参数无法存储到long变量中，将引发异常。实际情况确实如此：

```
(* 123 1234567890123456789)
```

上述代码将引发如下异常：ArithmeticException integer overflow clojure.lang.Numbers.throwIntOverflow。对于7.3.2节的表格中列出的所有算术运算函数，都有包含撇号（'）后缀的变种。这些变种在必要时将结果转换为BigInt实例：

```
(*' 123 1234567890123456789)
```

你可能会问，上述代码的结果是什么呢？为151851850485185185047N。

2. 字符串和字符

与Scala一样，Clojure也使用标准JVM类（java.lang.String）来表示字符串。Clojure的数学函数不能用于字符串。虽然很多语言都使用加法运算符（通常为+）来拼接字符串，但Clojure函数+不支持字符串，而必须使用函数str：

```
(str "Good" "night!")
```

上述表达式的结果为Goodnight!。

Clojure提供了大量专门用于字符串的函数，其中很多都是在命名空间clojure.string中声

明的。例如，要将列表转换为字符串，可使用clojure.string中的函数join：

```
(clojure.string/join "/" ["10", 20, 30M, 40N])
```

结果为10/20/30/40。参数指定了要在元素之间添加什么样的分隔符。虽然这个列表包含多种不同类型的值（字符串"10"、整数20、BigInt实例30M和BigDecimal实例40N），但这些值都被转换为字符串。

推荐使用require来导入clojure.string库：

```
(require '[clojure.string :as str])
(str/join, "/" [1, 2, 3])
```

下表列出了命名空间clojure.string中其他常用的函数，其中的示例假设使用前述代码行(require '[clojure.string :as str])导入了clojure.string库：

名　　称	描　　述	示例（输入→输出）
blank?	如果传入的字符串为nil、空或只包含空白字符，就返回true；否则返回false	(str/blank? " ")→true
capitalize	将字符串的第一个字符转换为大写，而其他字符都转换为小写	(str/capitalize "JVM rules") → "Jvm rules"
ends-with?	指出字符串是否以指定的字符或字符串结尾	(str/ends-with? "Hi" "i")→true
last-index-of	返回指定字符串最后一次出现的位置（从零开始的索引）；如果没有找到这样的字符串，就返回null	(str/last-index-of "HELLO" "L")→3
lower-case	将字符串中的字符都转换为小写，并返回结果	(str/lower-case "HeLlO")→"hello"
replace	将字符串中的子串替换为另一个子串	(str/replace "HELLO" "ELLO" "i!") s"Hi!"
reverse	将字符串中的字符按相反的顺序排列，并返回结果	(str/reverse "!iH")→"Hi!"
split	根据正则表达式对字符串进行拆分；请注意，在Clojure中，使用前缀#来表示正则表达式	(str/split "a-b-c" #"-") → ["a" "b" "c"]
split-lines	根据换行符（Windows中为CR＋LF；Linux和众多其他流行的操作系统中为LF n）对字符串进行拆分	(str/split-lines "A\nB\r\nC") → ["A" "B" "C"]
trim	删除开头和末尾的空白字符（即空格、制表符、CR和LF）	(str/trim " A\nBC\t\n")→"A\nBC"
upper-case	将字符串中所有的字符都转换为大写，并返回结果	(str/upper-case "abC")→"ABC"



除这里列出的函数外,这个库中还包含其他函数,请务必阅读有关这个库的文档。

Clojure不使用基本类型`char`来表示字符。在Clojure中,字符为`java.lang.Character`实例。要创建这样的实例，可在字符前加上反斜杠：

```
(println \H \e \l \l \o)
```

这将向控制台打印`H e l l o`。在这里，传递给函数`println`的参数不是字符串，因此不需要使用双引号将它们括起。相反,指定的每个字符都将转换为一个`java.lang.Character`实例,而函数`println`只是按从左到右的顺序打印这些实例。

如果你知道字符的UTF-16码点，也可使用函数`char`：

```
(println (char 65))
```

这将打印`A`，因为字符`A`的ASCII编码为65，而UTF-16编码与ASCII编码兼容。请注意，在这个示例中，里面的括号必不可少，因为需要先计算参数再将其传递给函数。如果你省略里面的括号，将打印指向函数`char`的引用和65。

3. 集合

与Scala一样，Clojure也提供了集合类实现。为提供不可修改和永久性集合，必须这样做（前面说过，这里的永久性指的是集合中的新数据副本使用指向既有数据副本的引用，以节省内存）。

Clojure为集合类提供了接口,使用类的哪个实现也由Clojure决定。调用特定的函数时,Clojure还可能决定修改实现。鉴于所有的集合都实现了相同的接口，这通常不是问题。

接下来将介绍如下集合类型：

- 列表；
- 向量；
- 集；
- 散列映射。

(1) 列表

常规列表是使用函数`list`创建的：

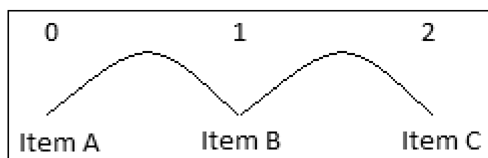
```
(list "item 1" "item 2")
```

列表实现了Clojure接口`ISeq`——所有Clojure集合都实现了这个接口。可对列表进行迭代（迭代将在本章后面讨论），但不同于稍后将讨论的向量，基于索引的列表迭代未经优化。Clojure提供了函数`nth`，这个函数迭代列表以找到指定的元素。由于需要遍历整个集合，因此这个函数的

效率不是很高：

```
(nth (list "item A" "item B" "item C") 2)
```

在这里，我们让函数nth遍历列表，以取回第3个元素（索引是从0开始的）。它像预期的那样返回item C。下面说明了nth的工作原理：



如果列表包含数百个元素，检索效率将极低。所幸稍后你将看到，Clojure提供了一种更适合通过索引来获取元素的数据结构。

要在既有列表中添加元素来创建新列表，可使用函数conj。下面是一个这样的示例：

```
(conj (list 10 20 30) 40 50)
```

令人意外的是，返回的新列表为(40 50 10 20 30)，这是因为列表经过了优化，将新元素放在开头。

(2) 向量

向量很像Java类ArrayList。与ArrayList对象一样（但不同于列表），向量针对根据索引获取元素进行了优化。要创建向量，可使用函数vector：

```
(vector 10 20 30)
```

上述代码返回[10 20 30]。你也可使用方括号来创建向量，因此下面的代码也管用：

```
[10 20 30]
```

在这个示例中，我们没有添加括号，因为如果这样做，就必须添加函数调用。

虽然可使用函数nth来获取指定索引处的元素，但不推荐这样做。前面讨论过，函数nth迭代整个集合来查找元素。对于向量，使用函数get的效果要好得多。函数get通过直接读取引用来获取元素，但不能用于列表：

```
(get [10 20 30] 1)
```

这个表达式返回20。

向量经过了优化，它将新元素添加到集合末尾。因此，下述在向量中添加元素的代码将返回[10 20 30 40 50]，这完全在意料之中：

```
(conj [10 20 30] 40 50)
```

(3) 集

集是一组各不相同的值，你不能再添加已有的值。要创建散列集（一种不按添加顺序存储元素的集），可使用下面的字面量表示法：

```
#{ 10 20 30 }
```

在返回的集中，元素的排列顺序可能与指定顺序不同；我执行这段代码时，返回的是`#{20 30 10}`。

要创建按添加顺序排列元素的有序集，可使用函数`sorted-set`：

```
(sorted-set 10 20 30)
```

在这个示例中，返回的集为`#{10 20 30}`，其中元素的排列顺序与指定顺序相同。需要指出的是，有序集的开销比散列集大。

与向量一样，要从集中取回值，可使用函数`get`；要在集中添加值，可使用函数`conj`。

(4) 散列映射

与向量类似，要创建散列映射，可使用函数`hash-map`，也可使用`{}`。下面的两段代码等价：

```
(hash-map :key1 "value1", :key2 "value2")
```

```
{:key1 "value1", :key2 "value2"}
```

将键–值对分隔开来的逗号是可选的，但建议不要省略它们，因为这样可提高代码的可读性。与大多数标准散列映射实现一样，键–值对的排列顺序是不可预测的。在我的计算机中，打印出来的输出如下：`{:key2 "value2", :key1 "value1"}`。

在前面的示例中，两个键都是关键字。要创建关键字，可在名称前加上冒号（:）。在散列映射中，键并非必须是关键字，但推荐尽可能使用关键字，因为它们的效率非常高，让查找的速度非常快。关键字的结果为其本身，因此当你在REPL中输入`:key1`时，将看到打印出来的输出也为`:key1`。

要获取散列映射中的值，可使用你熟悉的函数`get`，这个函数也用于获取向量中的元素：

```
(get { :key1 "value1", :key2 "value2" } :key2)1
```

这将返回`"value2"`。

对于将关键字用作键的散列映射，获取其中的值时可省略`get`函数调用：

```
(:key1 { :key1 "value1", :key2 "value2" })
```

这将返回`"value1"`。

要通过在既有散列映射中添加和修改键–值对来创建新的散列映射，可使用函数assoc：

```
(assoc { :k1 "v1", :k2 "v2" } :k3 "v3", :k2 nil)
```

这将创建如下散列映射：{:k1 "v1", :k2 nil, :k3 "v3"}。

要合并两个散列映射，可使用函数merge：

```
(merge { :k1 "v1", :k2 "v2" } { :k2, nil, :k3 "v3" })
```

这也将返回{:k1 "v1", :k2 nil, :k3 "v3"}。

要检查散列映射是否包含指定的键，可使用函数contains?：

```
(contains? { :k1 "v1", :k2 "v2" } :k3)
```

由于这里的映射没有包含键为关键字:k3的元素，因此上述代码返回false。

4. 数组迭代和循环

Clojure函数for的功能极其强大。在最简单的情形下，这个函数只是对集合进行迭代：

```
(for [x ["A" "B" "C"]]
 x)
```

这将返回新列表("A" "B" "C")。

可在for循环中包含多个迭代器：

```
(for [x [1 2 3],
      y [100 200]]
 (+ x y))
```

与往常一样，逗号是可选的。这些代码返回新列表(101 201 102 202 103 203)。

通过使用关键字:let，可定义一个局部函数，并在每次迭代中调用它。这个局部函数只能在for循环内使用，且不能修改：

```
(for [x [10 20 30]
      :let [y (* 2 x)]]
 (list x y))
```

这些代码返回一个包含三个列表的新列表：((10 20) (20 40) (30 60))。

还可使用关键字:while来添加一个返回true或false的函数；当这个指定的函数返回false时，将停止迭代：

```
(for [x [10 20 30]
      :let [y (* 2 x)]
      :while (<= y 40)]
 (list x y))
```

这些代码返回列表((10 20) (20 40))。

最后，可使用关键字:when来指定条件。指定的条件表达式必须返回true或false。如果这个表达式为true，就将相应的值添加到列表中，否则就忽略它，并继续迭代：

```
(for [x (range 10)
      :let [y (* x x)]
      :when (= (mod y 2) 0)]
y)
```

上述代码返回(0 4 16 36 64)，其中函数(range 10)生成一个包含0~9（含）的序列。

5. 条件

前面在函数for中指定关键字:while和:when时，简要地介绍了一些条件。同样，Clojure没有提供条件运算符，而是提供了一些结果为true或false的普通函数。下表列出了一些最重要的条件函数，它们的参数数量都是可变的，这意味着你至少可以指定两个参数。

函　数	描　述	示例（输入→输出）
==	如果指定的所有参数表示的值都相同，就返回true	(== 42 42.0 42M 42N)→true
not=	只要至少有一个参数与下一个参数不相等，就返回true	(not= 1 1 2)→true
<	如果传入的每个参数都比下一个参数小，就返回true	(< -1 5 10)→true
>	如果每个参数都比下一个参数大，就返回true	(> 10 5 -1)→true
<=	如果传入的每个参数都小于或等于下一个参数，就返回true	(<= 5 5 6)→true
>=	如果传入的每个参数都大于或等于下一个参数，就返回true	(>= 6 6 5)→true
and	逻辑与	(and (> 6 5) (< -1 10))→true
or	逻辑或	(or (== 3 10) (> 5 3))→true
not	逻辑非	(not (== 1 5))→true

Clojure还提供了逻辑函数if：

```
(if (< 100 10) "This is true" "This is completely false")
```

如果第一个参数为true，就计算并返回第二个参数，否则返回第三个参数。上述表达式的结果显然为"This is completely false"。对于函数if，有几点需要说明。

- 用于函数if时，nil相当于false。
- else部分（第三个参数）并非是必不可少的。如果没有指定第三个参数，且条件为false，则整个表达式的结果为nil。

7.4 使用 Java 类

你知道，Clojure并不是一种面向对象的语言。Clojure开发小组给Clojure添加了多项功能，以确保它能够使用Java类库和其他JVM库中的类以及创建类。

Clojure支持两种创建类实例的方式。一是使用new：

```
(def x (new java.util.ArrayList () ))
```

这里定义了一个指向ArrayList实例的变量。通过传递一个空的ArrayList ()方法，没有给构造函数传递任何参数。另一种创建实例的方式是在类名后面加上一个句点：

```
(def x (java.util.ArrayList. () ))
```

注意，在ArrayList后面加上了句点。从功能上说，这两种方式没有什么不同。



在Clojure程序中使用可变集合前一定要三思。由于Clojure是一种函数式编程语言，通常应尽可能使用Clojure的不可变集合。

要对对象实例调用方法，可在方法名前面加上句点，再指定对象实例以及方法的参数：

```
(.add x 10)
```

在这里，方法add返回true。要查看变量x的内容，只需在REPL中输入x并按回车，REPL将打印[10]。

要对同一个实例调用多个方法，一种便利的方式是使用doto宏：

```
(doto x (.add 20) (.add 30) (.add 40))
```

doto宏返回指定的对象实例，因此上述代码打印[10, 20, 30 40]。

要访问类属性（静态字段），可使用全限定类名、斜杠和成员名：

```
(java.lang.Integer/MAX_VALUE)
```

这将返回2147483647。

再举一个例子——使用java.lang.System类的静态字段out打印一条消息：

```
(.printf System/out "Hello %s!!n" (into-array String (list "world")))
```

输出类似于下面这样：#object[java.io.PrintStream 0x4ea5b703 "java.io.PrintStream@4ea5b703"]（在你的计算机上，结果可能与此不同）。

上述代码片段演示了多种技巧。

- ❑ Clojure默认导入了java.lang包，因此无需使用全限定类名java.lang.System。

- ❑ PrintStream类的方法printf的第二个参数是一个字符串数组（String[]），因此我们使用Clojure函数into-array将只包含一个元素的列表转换为字符串数组。
- ❑ 方法printf返回当前PrintStream对象，因此上述表达式的结果为前面显示的对象。

使用 deftype 和 defrecord 创建简单的 Java 类

Clojure提供了deftype和defrecord宏，你可使用它们来动态地生成包含构造函数和字段的Java类，而这些Java类可用来定义存储简单数据的类型。下面是一个deftype宏的使用示例：

```
(deftype Position2D [x y])
```

这里定义了一个名为Position2D的类，其中包含两个字段——x和y。要根据这个类实例化一个对象，可像下面这样做：

```
(def position (Position2D. 5 10))
```

这创建了一个名为position的变量，它指向的Position2D对象的字段x和y的值分别为5和10。请注意，类名Position2D后面的句点是必不可少的，这种语法在前面讨论过。

使用这两个宏定义类时，如果指定了一个或多个Java接口，也可给类添加方法，但仅限于接口中定义的方法。假设我们要实现一种可关闭的资源，即它实现了Java类库中的java.io.Closeable接口：

```
(deftype SomeCloseableResource []
  java.io.Closeable
  (close
   [this]
   (println "Closing resource...")))
```

java.io.Closeable接口只定义了一个方法——close()。这个方法不接受任何参数，但在Clojure中，必须定义包含当前实例引用的变量（在Java中，这个变量名为this）。



在实例方法中，Java自动定义变量this，但在Clojure中，你必须手动为每个方法定义这个变量。

下面的示例演示了如何使用方法close()：

```
(def resource (SomeCloseableResource.))
(.close resource)
```

这将向控制台打印文本Closing resource...。

defrecord宏在语法方面与deftype相同，但存在如下重要的不同之处。

- ❑ 使用defrecord定义的类包含一个永久性映射，这引入了可扩展的字段——可在这个映射中添加定义类时没有定义的键。

- 使用deftype定义的类可包含可修改的字段，而在使用defrecord定义的类中，字段都是不可修改的。

下面是一个使用defrecord定义的简单类；这个示例表明，使用defrecord定义的类实现了一个类似于映射的接口：

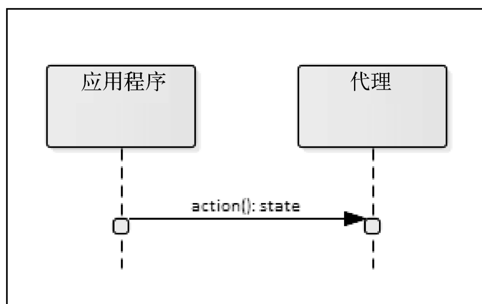
```
(defrecord Record [:field1 :field2])
(def rec (Record. "value1" "value2"))

(contains? rec :field2)
```

这个表达式的结果为true。

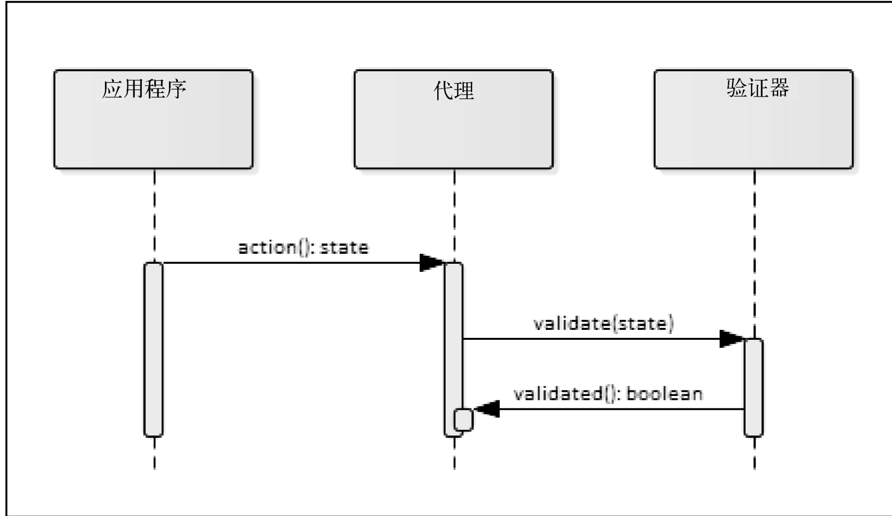
7.5 使用代理管理状态

为了在多线程程序中妥善地管理可修改的状态，Clojure提供了代理（agent）。每个代理都负责管理一个包含其状态的对象。在大多数情况下，状态对象都存储在不可修改的Clojure数据结构中。要修改特定代理的状态，可向它发送一个操作（action）。操作是普通的非阻塞函数，由代理执行；而操作的返回值将取代代理的当前状态。



代理运行在Clojure管理的一个内部线程池提供的线程中，它们可随时做出响应；处理操作时Clojure不会加锁。任何时候，其他代码都可安全地读取代理的状态，而不管这些代码运行在哪个线程中。操作是以异步方式发送给代理的，代理所在的线程收到操作后将执行它，并将代理的状态设置为操作的结果。

可给代理添加验证器（validator）。验证器是一个函数，对新状态进行验证，进而接受或拒绝。如果给代理添加了验证器，操作将首先发送给验证器；如果验证器接受了新状态，操作的响应将成为新的代理状态；如果验证器拒绝了操作，操作的返回值将被丢弃，并引发异常。



还可给代理添加监视器（watcher）。如果添加了监视器，将在成功地修改了代理的状态后调用它。引发异常后，代理将缓存错误，并停止接受操作，直到被重启。

由于代理运行在线程池提供的线程中，因此只要有代理在运行，JVM就不会关闭。为确保资源得以正确地释放，必须使用Clojure提供的函数来妥善地关闭代理。

代理示例

要创建代理，可使用函数agent。下面来创建一个存储发货单状态的虚构代理。为简单起见，我们只存储客户名称以及一个表明发货单是否已处理的布尔标志：

```
(def invoice-agent (agent (hash-map :name nil, :isProcessed false)))
```

这创建了一个代理，并将指向它的引用存储到变量invoice-agent中。为了存储这个代理的状态，我们使用了一个不可修改的映射，其中包含客户名称（最初为空）和一个指出发货单是否已处理的标志（最初为false）。

下面来给这个代理创建两个操作：

- ❑ 更新客户名称的操作update-customer-name；
- ❑ 更新发货单标志isProcessed的操作update-processed。

前面说过，操作是返回代理新状态的普通函数。这些操作将代理的当前状态作为输入，它们的代码如下：

```
(defn update-customer-name [state name] (assoc state :name name))
(defn update-processed [state flag] (assoc state :isProcessed flag))
```

这两个函数都自动从代理那里获取其当前状态，它们的第二个参数分别是新的客户名称以及表明发货单是否处理过的标志的新值。这两个函数都返回存储代理状态的散列映射的副本。有关散列映射和函数assoc，请参阅前面介绍散列映射的一节。这个返回值将成为代理的新状态。

下面通过设置客户名称来测试这个代理：

```
(send invoice-agent update-customer-name "Your Name")
```

函数send的第一个参数指定了要将操作发送给哪个代理实例，第二个参数是要发送的操作，而其他所有参数都是操作的参数［但操作的第一个参数（在这里，两个操作的第一个参数都是state）不是在这里指定的］。操作及其参数被发送给代理，但函数send在代理处理操作前就已返回。由于代理运行在独立的线程中，因此无法预测操作将在何时被处理。然而，由于这并不是一个有众多线程同时运行的繁忙应用程序，因此操作很可能在你按下回车后就能得到处理。函数send返回相应的代理实例，但由于我们有指向这个代理实例的引用（变量invoice-agent），因此忽略这个返回值。

要检查代理的当前状态，可输入指向它的变量并加上前缀@：

```
@invoice-agent
```

这个表达式的结果应为{:name "Your Name", :isProcessed false}。

这说明前述代码管用！下面来添加一个验证器，它在发货单处理过后拒绝将客户名称设置为nil的操作。验证器函数获取新的状态，并在这种修改可接受时返回true，在要拒绝修改时返回false：

```
(defn validator-invoice [state]
  (if
    (and
      (get state :isProcessed)
      (clojure.string/blank? (get state :name)))
    false
    true)
)
```

函数blank?来自clojure.string库，它在指定的字符串为nil时返回true，否则返回false。下面将这个验证器添加到代理中，为此可使用函数set-validator!：

```
(set-validator! invoice-agent validator-invoice)
```

接下来，将已处理标志改为true，过段时间后再查看当前状态：

```
(send invoice-agent update-processed true)
@invoice-agent
```

你将看到发货单已处理过：{:name "Your Name", :isProcessed true}。

下面尝试将客户名称设置为nil，看看这个验证器是否管用：

```
(send invoice-agent update-customer-name nil)
@invoice-agent
```

你将看到客户名称并没有被重置为nil，因为验证器禁止这样做。要查看代理是否出现了错误，可使用函数agent-error：

```
(agent-error invoice-agent)
```

操作被验证器拒绝时，代理系统将自动引发栈跟踪（traceback），因此上述代码返回的内容如下（为简洁起见做了删节）：

```
#error {
  :cause "Invalid reference state"
  ...
}
```

注意，除非重启，否则这个代理不会再接受新的操作。在此期间，收到的操作存储在缓冲区。要重启代理，只需调用函数restart-agent即可：

```
    (restart-agent invoice-agent (hash-map :name nil, :isProcessed false)
:clear-actions true)
```

重启代理时，必须指定其状态。要让代理对其在没有响应期间存储到缓冲区的操作进行处理，可在可选的关键字:clear-actions后面指定false。

要妥善地关闭代理系统，可执行函数shutdown-agents：

```
(shutdown-agents)
```

这样代理线程将停止，不再对操作做出响应。

7.6　风格指南

Clojure开发小组没有在官网发布编程风格指南，但有一个社区推动的风格指南文档（http://github.com/bbatsov/clojure-style-guide）。

这个文档讨论的一些重要规则如下。

- 表示缩进时，通常使用两个空格。
- 使用defn定义函数时，将函数名和输入参数放在同一行中，并让函数体重启一行：

```
(defn function1 [input]
( ...function calls here... )
```

- 对于一行放不下的参数，让其各行内容垂直对齐并使用一个空格：

```
(defn function2 []
  (str
```

```
"Hello"
" and goodbye"))
```

- 不要使用逗号来分隔列表的各个元素。
- 以得体的方式定义散列映射。在同一行放置多个键–值对时，使用逗号来分隔它们。
- 不要让右括号独占一行（在刚才的function2示例中，参数和函数块都到最后一行结束）。
- Linux换行符（LF）比Windows换行符（CR + LF）好。
- 每行包含的字符数最好不要超过80个。

7.7　小测验

(1) Clojure是纯粹的函数式编程语言吗？

a) 对，它是纯粹的函数式编程语言。
b) 不对，它是函数式编程语言，但不是纯粹的函数式编程语言，因为它允许修改状态。
c) 不对，但Clojure是纯粹的OOP语言。
d) 不对，Clojure不是函数式编程语言。

(2) 下面的代码能够在Clojure REPL中运行吗？

```
(10)
```

a) 能。这些代码合法，将返回10。
b) 不能，因为最后一个列表项必须是函数。
c) 不能，因为第一个列表项不是函数。
d) 以上答案都不对。

(3) 对于整数，Clojure默认将其视为哪种数据类型（如果这个整数在该数据类型的可能取值范围内）？

a) 基本类型long。
b) 包装类java.lang.Long的实例。
c) 基本类型int。
d) 包装类java.lang.Integer的实例。

(4) 下面的程序的输出是什么？

```
(println + 25 25)
```

a) 它打印50。
b) 它打印指向函数+的引用以及"25 25"。
c) 引发异常。

d) 以上答案都不对。

(5) 如果你需要一个可随机访问其元素的可迭代集合，该选择使用列表还是向量?

a) 向量。
b) 列表。
c) 向量和列表都可以。
d) 以上答案都不对。

7.8　小结

本章介绍了Clojure语言，它与本书介绍的其他语言都大不相同。在REPL环境中编写了大量的单行表达式后，但愿你已发现Clojure语法学习起来一点都不难。仅通过创建包含表达式的列表（表达式通常是嵌套在多个列表中），就可编写出可读性极强的代码。我们还了解到，Clojure是一种函数式编程语言，其重要的数据结构大多是不可修改的。不同于Java，Clojure从本质上说并不是面向对象的语言，但能够很好地与JVM平台兼容。我们创建了一些JVM对象实例、调用了它们的方法并读取了它们的字段。最后，我们介绍了代理，这是一种在多线程应用程序中安全地管理状态的方式；我们还编写了一个应用程序来尝试使用代理。

至此，你熟悉了最重要的Clojure规则，可编写货真价实的应用程序了。

第 8 章

Clojure编程



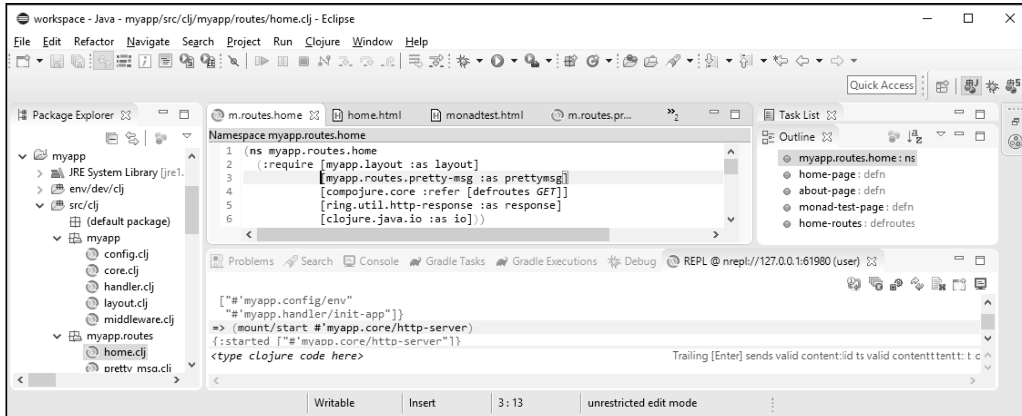
前一章介绍了如何直接在REPL中输入代码来编写Clojure程序。这虽然可行，但即便是较小的项目，也必须将代码放在多个文件中。本章的重点是开发项目，为此我们也将使用Eclipse IDE来编写代码，这都是拜插件Counterclockwise所赐，它让Eclipse IDE支持Clojure。开发Clojure项目时，最常用的构建工具是Leiningen，本章将大量使用它。

我们将创建两个项目，其中一个项目的重点是函数式编程语言大量使用的monad，我们将以测试驱动开发的方式探索这个主题。我们还将使用流行的Clojure微Web框架Luminus来创建一个非常简单的Web应用程序。本章将介绍的主题如下：

- Eclipse IDE插件Counterclockwise；
- 构建工具Leiningen；
- 使用Clojure创建可执行的程序；
- 在Eclipse IDE中创建Counterclockwise项目；
- 以测试驱动开发的方式探索monad；
- Web框架Luminus。

8.1 Eclipse IDE 插件 Counterclockwise

要让Eclipse IDE支持Clojure，必须安装一个插件——Counterclockwise。也有独立版Counterclockwise，它不要求你已安装Eclipse IDE。与本书介绍的其他语言一样，本章也将提供Counterclockwise插件版安装指南。通过安装插件，可让一个Eclipse IDE实例支持所有的语言。



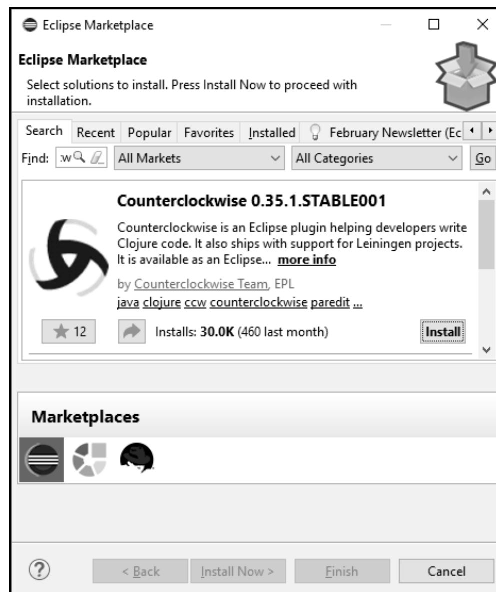
建议你阅读Counterclockwise文档（http://doc.ccw-ide.org）。

8.1.1　安装插件 Counterclockwise

我们将使用Eclipse Marketplace提供的Counterclockwise版本。为此，请在Eclipse IDE中按如下步骤安装它。

(1) 在Eclipse IDE中，选择菜单Help>Eclipse Marketplace...。

(2) 在搜索框中输入counterclockwise并按回车。找到Counterclockwise小组开发的最新版Counterclockwise，并单击相应的Install按钮。



(3) 如果你同意许可条款，接受它们并单击Finish按钮。相比于本书使用的其他插件，这个插件的安装时间可能更长。与独立小组开发的众多其他插件一样，Counterclockwise安装文件当前也是没有签名的。这导致Eclipse IDE会显示安全警告，请单击OK按钮确认要安装。

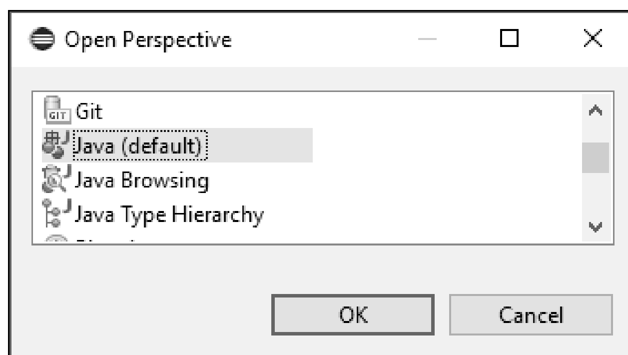
(4) Eclipse IDE询问是否要重启时，单击Yes按钮。

8.1.2 切换到 Java 透视图

不同于众多其他的语言插件，Counterclockwise插件不会在Eclipse IDE中添加专用的Clojure透视图，而要求开发人员激活Java透视图。请在Eclipse IDE窗口的右上角找到并单击：



如果看不到这个按钮，请单击工具栏中的Open Perspective按钮，这将打开一个对话框，其中包含可用的透视图，然后找到并单击Java透视图：





每当你需要进行Clojure编程时或激活其他透视图时，都需要重复这个步骤。

在本章后面将看到，当你新建Counterclockwise项目或打开既有的Counterclockwise项目时，Counterclockwise也会在Java透视图的用户界面中添加多个与Clojure相关的元素。

8.2 构建工具 Leiningen

对Clojure开发来说，Leiningen是事实上的标准构建工具。这个项目的宣传语是这样说的：

> 用于自动化Clojure项目，让你不再火烧眉毛。

Counterclockwise插件自带Leiningen，但编写本书期间，这个捆绑版本并不是最新的，因此推荐你手动安装最新版。我们将演示如何配置Counterclockwise，使其使用你手动安装的Leiningen。

有关这方面的更详细信息，请参阅Leiningen网站（http://leiningen.org）。

安装 Leiningen

Leiningen的安装步骤非常简单。请访问这个项目的网站，并找到Install部分，然后从这里下载用于Linux和macOS或Windows的安装脚本，再按如下步骤来运行这个脚本。

(1) 在命令提示符（Windows）或终端（macOS/Linux）中，切换到下载的脚本所在的目录。

(2) 执行这个脚本（在Linux/macOS中，必须使用命令`chmod +x SCRIPTNAME`添加执行权限）。你将看到如下消息，指出在你的主目录中找不到JAR文件：

```
C:\Users\USERNAME\.lein\self-installs\leiningen-2.7.1-standalone.jar
    can not be found.
You can try running "lein self-install"
or change LEIN_JAR environment variable
or edit lein.bat to set appropriate LEIN_JAR path.
```

(3) 执行命令`lein self-install`。它将下载一个JAR文件，并将其放到前述消息指出的目录中。

(4) 执行命令`lein`，它将显示一个选项列表。

(5) 将脚本所在的目录添加到环境变量`Path`中，或将其复制到该环境包含的某个目录中。

切换到不同的目录，并执行命令`lein repl`来检查安装情况，这将启动Clojure的REPL。输入任何Clojure表达式，如：

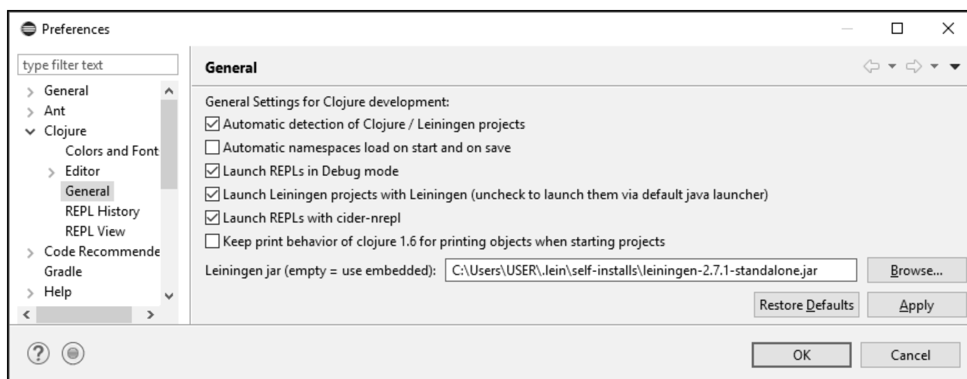
```
(+ 1 2)
```

你将在控制台中看到`3`。Leiningen给REPL添加了命令`exit`，因此现在可输入`exit`并按回车来退出REPL。这样做时，你将在控制台中看到令人振奋的消息“Bye for now!”。

配置Counterclockwise，使其使用你手动安装的Leiningen。

(1) 在Eclipse IDE中，选择菜单Window>Preferences。

(2) 在打开的Preferences对话框左边，选择Clojure>General，再单击Leningen jar (empty = use embedded)：旁边的Browse...按钮。安装脚本lein将Leiningen安装在你的用户主目录下的一个名为.lein的目录中（请注意前缀句点），而JAR文件位于这个目录下的子目录self-installs中。在我的系统中，这个文件名为leiningen-2.7.1-standalone.jar：



(3) 选择这个文件，再单击OK按钮关闭Preferences对话框。

(4) 手动重启Eclipse IDE让修改生效。

8.3 创建可执行的 Clojure 程序

到目前为止，我们都是在Clojure交互式REPL shell中输入代码片段。前一章说过，Clojure没有自带独立的编译器。要创建可执行的Clojure程序，必须在代码中调用一个Clojure宏，让内置编译器生成JVM类文件。这个宏仅在Clojure编译代码时生成类文件，但在你执行已编译的代码时什么都不做。

8.3.1 在不使用 Leiningen 的情况下将代码编译成类文件

我们先在不使用构建工具的情况下创建一个可执行的类，了解差别后，你将更清楚Leiningen的好处。我们使用普通文本编辑器（而不是Eclipse IDE）来创建这个小型项目。创建根目录testproject1，用于存储示例文件，再在这个目录下创建如下必不可少的子目录：

- com；
- com\example；
- classes。

要让Clojure将类文件写入这些目录，你必须生成JVM类。为此，一种选择是定义一个命名空间，并指定关键字:gen-class。请将如下源代码文件保存到目录com\example并将其命名为main.clj：

```
(ns com.example.main
  (:gen-class :name com.example.Main))
  (defn -main [] (println "hi!"))
  (compile 'com.example.main)
```

命名空间是通过函数ns的第一个参数指定的，仅供Clojure使用，但最好让它与JVM包名一致。

请注意，在Clojure中，一种最佳实践是包名全部小写。Clojure脚本的文件名和目录结构必须与Clojure命名空间匹配，这很重要。这就像Java要求源代码目录结构与包名匹配一样。因此，文件main.clj必须存储在目录com\example中。将关键字`:gen-class`和`:name`作为参数传递给了函数`ns`，其中`:name`指定了JVM类的全限定名。为遵循JVM约定，将类名指定为了`Main`。

接下来，在这个类中添加了方法`main`。通过在方法名前面加上`-`，告诉Clojure应将它加入到`com.example.Main`类中。实际上，也可使用其他的前缀，这为在一个文件中定义多个类提供了极大的便利。Clojure内置了将方法`main`编译为JVM变种`static void main(String[] args)`的逻辑。这个方法只是打印一条问候消息。

最后，调用了宏`compile`。调用这个宏时，必须通过参数指定要编译的Clojure命名空间。注意，指定命名空间时，在它前面加上了一个单引号，且没有与之匹配的单引号；这并非输入错误。在前面加上`'`后，名称将变成符号。对于符号，不会立即对其进行计算，而是直接将其传递给函数或宏。

要编译这些代码，在命令提示符或终端中确保位于项目的根目录（包含子目录classes和com的目录）下，再执行如下命令：

```
java -cp ".;.\classes;c:\PATHTO\clojure\clojure-1.8.0.jar" clojure.main
com\example\main.clj
```

请将`PATHTO\clojure`替换为Clojure JAR文件的路径，并将版本号替换为你安装的版本。如果你使用的是Linux或OS X，还需将类路径目录分隔符`;`替换为`:`。

Clojure将编译这些代码，并将生成的类文件写入到目录classes中。目录classes必须包含在类路径中，否则编译器将崩溃。如果你查看目录classes，将发现Clojure生成了多个类文件，这是因为Clojure内部基础设施需要一些支持类。你无需关心这些支持类，只需确保在运行应用程序时在类路径中包含它们即可。该来运行这个应用程序了。为此，切换到目录classes并执行如下命令：

```
java -cp ".;c:\PATHTO\clojure\clojure-1.8.0.jar" com.example.Main
```

对于这个命令中的Clojure安装路径和目录分隔符，像前面的Java命令一样进行处理。你将看到消息“hi!”。

这个过程之所以复杂，主要是因为需要设置类路径。下一节你将看到，Leiningen让这个过程简单得多。

8.3.2　使用 Leiningen 编译项目

这次我们让构建工具Leiningen负责处理所有的细节。我们先让Leiningen为应用程序创建空的项目框架，再编译并运行它。

我们首先来新建一个用作项目根目录的目录并确保它为当前目录。为此，执行下面的命令，它根据Leiningen提供的一个模板生成一个空的项目框架：

```
lein new app testproject2
```

Leiningen将新建目录testproject2，其中包含基于模板app创建的项目文件。Leiningen还提供了其他模板，如用于创建库的模板（没有指定模板时，默认将使用它）。你还可以创建自定义模板。

请查看生成的目录的内容，其中有可用于存储文档的doc文件。还有用于存储项目源代码文件的目录src/testproject2，这个目录包含文件core.clj，其中包含类似于Hello World的脚本的代码。其他目录包含test和target，分别用于存储单元测试和编译后的文件。在项目的根目录中，有一个名为project.clj的文件，其中包含Leiningen用来构建项目的构建文件，后面将更详细地研究它。

确保当前目录为项目根目录（包含文件project.clj的目录），并执行如下命令来运行项目：

```
lein run
```

这将显示“Hello, World”。如你所见，使用Leiningen来运行项目一点都不麻烦，根本不需要手动指定复杂的类路径。最后，我们来编译这个项目。模板app配置了便利的任务uberjar，这个任务不仅将代码编译成类文件，还将它们放到JAR文件中。同样，确保当前目录为项目根目录，并执行如下命令：

```
lein uberjar
```

这将创建子目录target/uberjar，其中有两个jar文件，还有包含类文件的子目录classes。一个JAR文件较大，名为testproject-0.1.0-SNAPSHOT-standalone.jar，而另一个则小得多，名为testproject-0.1.0-SNAPSHOT.jar。差别在于较小的版本不包含Clojure运行时库，而较大的版本是个独立的JAR文件，包含所有必要的依赖。请切换到目录target/uberjar，并看看较大的版本能否运行：

```
java -jar testproject-0.1.0-SNAPSHOT-standalone.jar
```

你将再次看到问候消息。

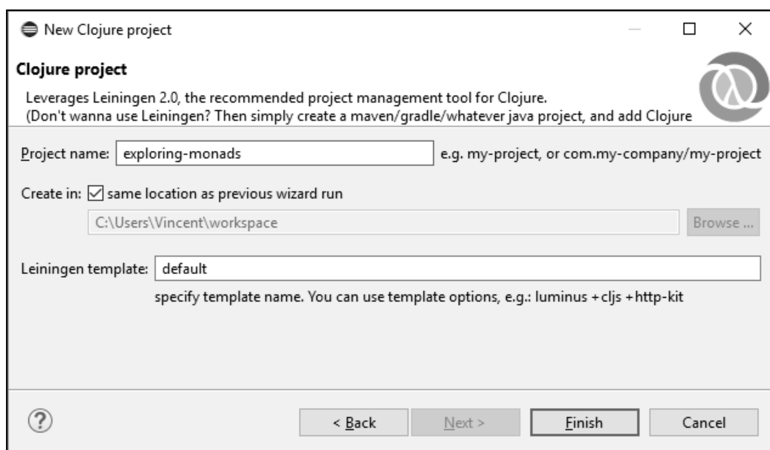
有趣的是，注意源代码文件src/testproject2/core.clj并不包含对Clojure函数`compile`的调用。使用Leiningen的编译任务时，无需调用函数`compile`，因为Leiningen会在内部处理编译任务。

8.4 新建 Counterclockwise 项目

尝试使用过Leiningen后，就可在Eclipse IDE中使用Counterclockwise插件创建第一个项目了。

(1) 在Eclipse IDE中，右击Package Explorer的空白处并选择New>Other...。
(2) 在打开的Select a wizard对话框中，选择Clojure>Clojure Project并单击Next按钮。

(3) 将项目名设置为exploring-monads，并确保选择了Leiningen模板default：



(4) 单击Finish按钮生成项目。



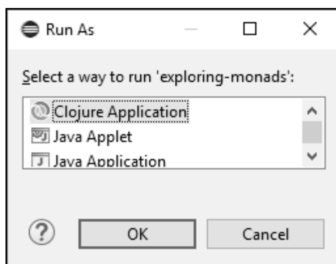
注意，在这里可选择Leiningen支持的其他任何模板，但需要注意的是，生成项目时使用的是Counterclockwise内置的Leiningen版本，而这种版本可能不是最新的。你将在本章后面看到，对于这个问题，可通过从命令行创建项目来解决。

项目创建过程需要一段时间。创建完毕后，请在Package Explorer中展开创建的项目，以查看其内容。你应对其目录结构很熟悉。

打开文件src/exploring_monads/core.clj，并在末尾添加如下代码行：

```
(foo "I'm tired of hearing: ")
```

为检查Counterclockwise的安装情况，单击Package Explorer中的项目名，再单击Eclipse IDE工具栏中的Run按钮。然后将出现一个对话框，让你选择运行配置；请选择Clojure application并单击OK按钮：

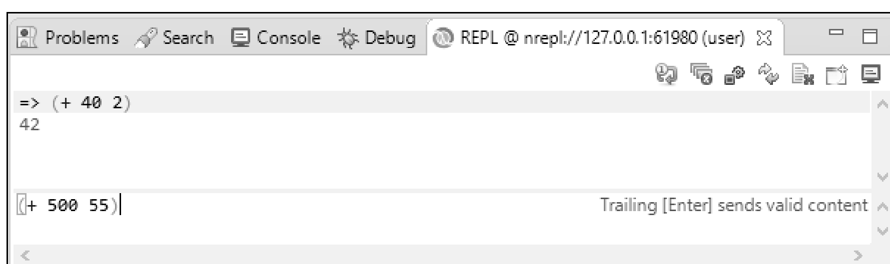


Counterclockwise将加载Clojure REPL，这可能需要一段时间。加载完毕后，将新增一个包含

REPL的选项卡。在Eclipse IDE中运行REPL后，就可运行代码了：在Package Explorer中单击文件core.clj，再单击工具栏中的Run按钮。你将在控制台中看到一条非常粗鲁的消息："I'm tired of hearing: Hello, World!"

8.4.1 Eclipse IDE 中的 Clojure REPL

项目的Clojure源代码是在运行在Eclipse中的Clojure REPL实例中运行的。REPL窗口包含两个窗格，上面的窗格包含REPL的输出，而下面的窗格可用于输入命令。当你在下面的窗格中输入命令并按回车时，命令将出现在上面的窗格中，然后命令被执行并显示其输出：



你可同时运行多个REPL实例。要再运行一个REPL实例，可单击项目名，再单击工具栏中的Run按钮，并在Eclipse让你选择运行配置时选择Clojure Application。Counterclockwise将询问你是想再启动一个REPL实例，还是使用原来的实例来运行项目的脚本。要启动新实例，可单击OK；要在原来的实例中运行脚本，可单击Cancel。

REPL处于最后一次激活的命名空间中。要在REPL中激活编辑器的当前命名空间，可按Ctrl + Alt + N（在macOS中为cmd + alt + N）。

8



在REPL中输入命令时，如果出现一条消息，指出命令在当前上下文中找不到，可尝试按组合键Ctrl + Alt + N（在macOS中为cmd + alt + N）来切换命名空间。

8.4.2 更新项目的 Clojure 版本

选择的默认Clojure版本可能不是最新的。由于我们在Counterclockwise中选择使用了最新的Leiningen版本，因此可以更新Clojure版本。为此，在Package Explorer中打开构建文件project.clj。务必要打开project.clj，而不是同一个目录下包含Eclipse项目文件定义的.project文件。前面说过，project.clj是Leiningen构建文件，用于构建和运行项目。在我的系统中，这个文件类似于下面这样：

```
(defproject exploring-monads "0.1.0-SNAPSHOT"
  description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
```

```
          :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.6.0"]]
  :main ^:skip-aot exploring-monads.core
  :target-path "target/%s"
  :profiles {:uberjar {:aot :all}})
```

从中可知，Leiningen使用Clojure源代码来定义项目及其构建需求。这与Maven等构建工具不同，它们使用XML文件来存储这种信息。实际上，这淋漓尽致地展示了Clojure的**代码即数据、数据即代码**原则。这个构建文件是使用Leiningen宏`defproject`定义的，这个宏将一些特定的关键字（以冒号打头的单词）及其值作为参数。

`defproject`宏的参数包含大量元数据，如项目描述、项目URL和许可证等。`:main`键指出程序运行时调用的函数`main`是在命名空间`exploring-monads.core`中定义的，这个命名空间对应于子目录src/exploringmonads中的文件core.clj。`:dependencies`键指定了当前依赖。当前，这个项目唯一的依赖项是Clojure 1.6。

`:aot`引用处理的是Clojure的**预先**（ahead of time，AOT）编译功能。它决定了使用编译任务时，将把哪些命名空间编译到类文件中。如果将值设置为`:all`，就意味着将编译所有命名空间，对任务`uberjar`来说，这是合适的。对于`main`函数，通常忽略AOT功能，为此可指定元数据设置`^:skip-aot`。

为更新这个项目将使用的Clojure版本，请执行如下操作。

(1) 修改:dependencies部分的版本号。本书出版时，最新的Clojure版本为1.8.0，因此我将1.6.0改为1.8.0。
(2) 按Ctrl + S（在macOS中为cmd + S）将文件存盘。Counterclockwise将立即更新项目，并将Clojure版本替换为你指定的版本。
(3) 单击REPL选项卡的Close图标将其关闭。再次打开文件core.clj，并单击工具栏中的Run图标，你将看到新指定的Clojure版本支持的REPL选项卡。

8.4.3 添加依赖

我们需要使用`monads`库。请访问这个项目的主页https://github.com/clojure/algo.monads，以了解最新版本以及必须向Leiningen提供的依赖信息。

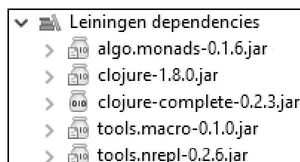
编写本书期间，最新版为0.1.6，而Leiningen依赖信息可在`monads`的项目主页中找到。我看到的情况如下：

```
[org.clojure/algo.monads "0.1.6"]
```

请在Eclipse中打开文件build.clj，并添加这个新依赖：在`:dependencies`向量中添加上述向量。添加这个依赖后，`:dependencies`行应类似于下面这样：

```
:dependencies [[org.clojure/clojure "1.8.0"],
               [org.clojure/algo.monads "0.1.6"]]
```

按Ctrl + S将文件存盘，Counterclockwise将立即更新项目。此时如果你在Package Explorer中展开条目Leiningen dependencies，将看到其中包含`algo.monads`库：



8.5　以测试驱动开发的方式探索 monad

在函数式编程中，monad用于创建简单的组件——以安全的方式串接一系列操作。每个组件都封装了一个值，并确保接下来将调用的组件能够将其输出作为输入。例如，如果组件A的输出为`nil`（`null`），而链条中的下一个组件不能将`nil`作为输入，整个链条的计算将自动停止。

在纯粹的函数式编程语言Haskell中，大量地使用了monad，在其他函数式编程语言中，monad也有用武之地。这里将简要地介绍monad，但不涉及复杂的理论和背景知识。

我们将创建一个简单的monad，它返回一条格式良好的消息：在传入的字符串前后添加大量的星号。请打开文件src/exploring-monads/core.clj，将其中的代码替换为下面的代码（我们将对这些代码进行单元测试）。函数`pretty-msg`的初始实现很简单，这让我们能够在提交这个API前检查它是否正确：

```
(ns exploring-monads.core)
(use 'clojure.algo.monads)
(defn pretty-msg [msg asterisk-amount]
   (str ""))
```

将这个文件存盘。下面来定义用于存储单元测试的源代码文件。为此，请打开文件test/exploring-monads/core_test.clj，并用下面的代码替换其既有内容：

```
(ns exploring-monads.core-test
  (:require [clojure.test :refer :all]
            [exploring-monads.core :refer :all]))

(deftest test-sane-parameters
  (testing "pretty-msg with with sane parameters"
    (is (= (pretty-msg "test" 3) "***test***"))))
```

在这里，我们定义了用于包含测试用例的命名空间`exploring-monads.core-test`。关键字`:require keyword`添加了到名称空间`clojure.test`和`exploring-monads.core`的引用。最后，我们定义了第一个测试用例，运行测试后将更详细地介绍它。



要将文件的内容发送给正在运行的Clojure REPL实例，可使用组合键Ctrl + Alt + S（Windows/Linux）或cmd + alt + s（macOS）。

为运行修改后的代码，请执行如下操作。

(1) 打开文件src/exploring-monads/core.clj并按Ctrl + Alt + S（在macOS中为cmd + alt + S）。
(2) 打开文件test/exploring-monads/core_test.clj并再次按Ctrl + Alt + S（在macOS中为cmd + alt + S）。
(3) 按Ctrl + Alt + N（在macOS中为cmd + alt + n）切换REPL选项卡的活动命名空间。

至此，运行的Clojure实例便有了编译后的代码。大多数IDE都支持单元测试，但Counterclockwise不支持。为了运行单元测试，一种办法是在REPL中手动执行函数`run-tests`，为此请在Clojure REPL选项卡中输入如下代码并按回车：

```
(run-tests)
```



为方便起见，可在脚本中调用`run-tests`，这样当REPL执行脚本时将自动运行这个函数。但不推荐这样做，因为当你使用Leiningen内置的测试命令时，这会导致冲突。

这将打印如下输出：

```
Testing exploring-monads.core-test
FAIL in (test-sane-parameters) (core_test.clj:7)
pretty-msg with with sane parameters
expected: (= (pretty-msg "test" 3) "***test***")
  actual: (not (= "" "***test***"))

Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
{:test 1,
 :pass 0,
 :fail 1,
 :error 0,
 :type :summary}
```

输出行`actual: (not(= "" "***test***"))`指出了问题所在：函数`pretty-msg`返回的是一个空字符串（`""`），而不是期望的字符串`**test***`。但我们对这个API（函数`pretty-msg`）很满意，因此可着手实现它，让这个测试通过。为此我们将使用一个monad。请打开文件src/monad_test/core.clj，并将其中的函数`pretty-msg`替换为如下代码：

```
(defn pretty-msg [msg asterisk-amount]
  (domonad identity-m
          [a asterisk-amount
           b (clojure.string/join (repeat a "*"))
           c (str b msg)]
          (str c b)))
```

根据定义，monad包含一个bind函数。这个bind函数确保一个组件的输出可用作下一个组件的输入，还可用来做决策（稍后你将看到这一点）。在这个示例中，我们只使用clojure.algo.monads库提供的预制monad类型，它们有内置的bind函数。在上述代码中，我们使用的monad类型identity-m确实有bind函数，但它没有使用bind函数来处理值或根据值做决策。后面我们将使用另一种利用了其bind函数的monad类型，并借此机会详细地阐述bind函数的原理。

将这个文件存盘，并按Ctrl + F11运行它。现在该来再次运行测试了。为此，请打开文件test/monad_test/core_test.clj，按Ctrl + Alt + S并在REPL中再次运行函数(run-tests)。情况应该比前一次测试更好：

```
Testing exploring-monads.core-test
Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
{:test 1, :pass 1, :fail 0, :error 0, :type :summary}
```

确定这个monad像预期那样工作后，下面来详细研究文件src/monad_test/core.clj中的代码。

(1) 首先，我们导入了monads库。
(2) 我们创建了一个函数，它接受两个参数：msg和asterisk-amount，其中前者包含消息，而后者指定要在消息前后分别添加多少个星号。
(3) 在函数体内，调用了domonad宏，并指定了monad类型identity-m——这种类型将在后面更详细地介绍。
(4) 我们定义了一个向量，用于包含monad的组件。
(5) 第一个组件被绑定到局部名称（local）a，并与一个整数相关联，而这个整数表示将在消息前后分别添加多少个星号。
(6) 第二个组件被绑定到局部名称b，并与接下来的表达式相关联。例如，如果a为3，b将为字符串***。
(7) 接下来指定了第三个组件，其值将被绑定到c。这个组件将b的结果与包含消息的参数msg合并。这是最后一个组件，因此向量到此结束。如果msg包含test，而a被绑定到3，那么c将包含***test。
(8) 最后，将c和b合并，得到整个monad的结果。在这个示例中，c包含***test，而b包含***。

因此，在单元测试中，pretty-msg返回的字符串为***test***。

根据上述代码，可得出如下几个结论。

- 在monad中，每个组件都可使用前一个组件的值。
- domonad宏的第一个参数为monad类型，这里为内置类型identity-m。稍后将介绍其他的monad类型。
- 第二个参数是一个包含组件的向量，其中每个组件的结果都被绑定到一个局部名称。

- 第三个参数是一个表达式。如果成功地执行了整个组件链，这个表达式的结果就是monad的返回值。

下面来做个试验：如果将要添加的星号数指定为nil，结果将如何呢？我们希望返回的结果为nil。为找出这个问题的答案，在测试脚本test/monad_test/core_test.clj中再添加一个测试用例：

```
(deftest test-nil-amount
  (testing "pretty-msg with with amount=nil"
  (is (= (pretty-msg "JVM" nil) nil))))
```

再次运行这个测试脚本：按Ctrl + Alt + S（在macOS中为cmd + alt + S，并在REPL中运行(run-tests)。你将看到有一个测试导致了错误（为简洁起见做了删节）：

```
ERROR in (test-nil-amount) (RT.java:1241)
pretty-msg with with amount=nil
expected: (= (pretty-msg "JVM" nil) "***JVM***")
actual: java.lang.NullPointerException: null
```

之所以会出现这样的错误，是因为monad的第二个组件调用的函数repeat不能接受nil（在Java中为null），因此引发NullPointerException异常。当前使用的monad类型identity-m接受所有的值，并假定下一个组件能够使用前一个组件的值。有一种内置的monad类型是maybe-m，它在一个组件的结果为nil时停止执行组件链。

先来介绍一些背景知识。前面说过，所有monad都有一个bind函数，这个bind函数可用来对前一个组件的输出数据进行转换，让下一个组件能够使用它。然而，它还可以用来根据返回值做出特定的选择。bind函数会被库自动调用。虽然前面使用的monad类型identity-m有bind函数，但它没有以任何方式对数据进行处理，也没有根据值来做出决策。另一方面，monad类型maybe-m有一个定义如下的bind函数（你无需输入这个定义，它是由我们使用的clojure.algo.monads库提供的）：

```
...
fn m-bind-maybe [mv f] (when-not (nil? mv) (f mv))
...
```

上述代码嵌套在一个这里没有显示的列表中。请不要过度关注定义函数的fn宏。组件执行完毕后，clojure.algo.monads会调用函数m-bind-maybe。这个函数被调用时，第一个参数(mv)将包含前一个组件的输出值，而第二个参数（f）是一个函数，包含将调用的下一个组件的实现。从上述代码可知，仅当mv不是nil时，才会调用（函数f表示的）下一个组件。因此，如果一个组件返回nil，将停止执行组件链。虽然monad的bind函数通常用于转换数据，但在monad类型maybe-m中，这个函数用于根据前一个组件的输出值判断是否可调用下一个组件。这也是bind函数的合法用途。

在文件src/monad_test/core.clj中，将monad类型从identity-m改为maybe-m。修改后，函数pretty-msg应类似于下面这样：

```
(defn pretty-msg [msg asterisk-amount]
  (domonad maybe-m
           [a asterisk-amount
            b (clojure.string/join (repeat a "*"))
            c (str b msg)]
           (str c b)))
```

将这个文件存盘，并按Ctrl＋F11再次运行主程序。由于我们没有修改测试代码，因此无需将单元测试代码发送给REPL，而只需在REPL中执行表达式(run-tests)即可。这次没有引发异常，而monad返回的结果为nil，因此测试通过了。只要有组件的输出为nil，monad类型maybe-m就会退出组件链。在这种情况下，返回的结果为nil。因此，现在clojure-test的结果如下：

```
Ran 2 tests containing 2 assertions.
0 failures, 0 errors.
```

可在monad中添加显式的条件，为此可在组件向量中添加关键字:when。例如，要在传入的消息不是至少包含一个字符的字符串时停止执行，可在组件向量末尾添加如下条件：

```
(defn pretty-msg [msg asterisk-amount]
  (domonad maybe-m
           [a asterisk-amount
            b (clojure.string/join (repeat a "*"))
            c (str b msg)
            :when (> (count msg) 0)]
           (str c b)))
```

如果这个条件为true，将正常执行；否则monad将停止执行并返回nil。这种条件可添加到所有类型的monad中，它将在计算结果表达式前被评估。

结束对monad的讨论前，必须指出的是，根据定义，monad还有一个unit函数（通常被称为return或result函数）。这个函数相当于类中接受输入参数的构造函数。unit函数初始化monad，并确保传入的数据可供第一个组件使用（这通常是通过对数据进行转换实现的）。与bind函数一样，unit函数也是由clojure.algo.monads库提供的，但在monad类型identity-m和maybe-m中，unit函数都没有以任何方式对传入的输入值进行处理。

通过创建自定义monad，可巧妙地利用unit和bind函数以及monad结束时返回的表达式，从而创建出能够轻松地与其他monad组合（串接）的monad类型。

8.6 Web 框架 Luminus

Luminus是一个微框架，用于快速开发功能强大的Clojure Web应用程序。它是可全面配置的，提供了强大的数据库支持，无论是传统SQL还是NoSQL数据库。这个框架易于上手，使用内置的Leiningen模板时尤其如此。强烈建议你在鼓捣Luminus的同时参阅其文档（http://www.luminusweb.net）。

在接下来的几节中，我们将使用模板myapp新建一个项目，再对其进行运行和探索。

8.6.1　创建 Luminus 项目

正如你见到的，Counterclockwise能够根据Leiningen模板生成项目，但存在一个问题：创建项目时，Counterclockwise使用其内置的Leiningen版本，但这个版本可能不是最新的。编写本书期间，根据`luminus myapp`模板生成项目时Counterclockwise会引发异常。为解决这个问题，可使用最新的Leiningen版本创建一个项目，再手动将其导入到Counterclockwise中。本章就将这样做。

在命令提示符（终端）中，切换到Eclipse workspace目录，并执行如下命令：

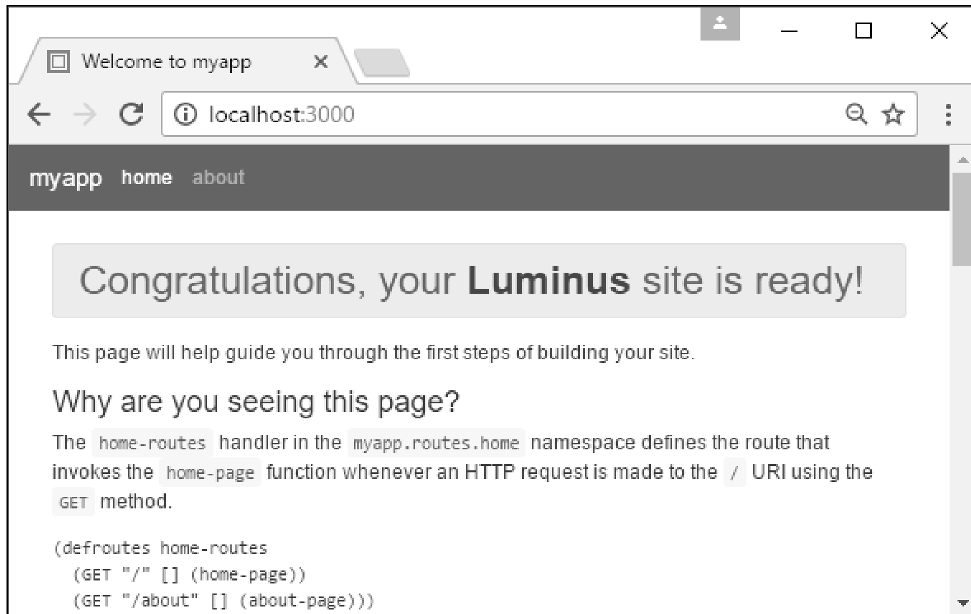
```
lein new luminus myapp
```

再切换到目录myapp，并执行如下命令：

```
lein run
```

首次运行这个项目时，Leiningen将下载必要的依赖。过段时间后，你将看到一条消息，指出即将启动HTTP服务器。Luminus内置的HTTP服务器使用的默认端口为3000。请使用你喜欢的浏览器访问这个Web应用程序，其地址如下：http://localhost:3000。

你将看到类似于下面的屏幕：



请仔细阅读这个初始页面中的文字——这个页面详尽地介绍了框架Luminus。另外，请单击链接about。

8.6.2 将项目导入 Counterclockwise

要导入这个Leiningen项目，并生成一个与Counterclockwise兼容的Eclipse IDE项目，请执行如下操作。

- 在Eclipse IDE中，右击Package Explorer的空白区域并选择Import...。在出现的Import对话框中，选择Projects from Folder or Archive，再单击Next按钮。
- 在出现的对话框中，单击文本框Import source旁边的Directory...按钮。切换到workspace目录（通常位于你的用户主目录中），并选择子目录myapp。确保选中了复选框Search for nested projects和Detect and configure project natures，再单击Finish按钮。

Eclipse IDE将发现这个项目与Counterclockwise兼容，进而生成相应的Eclipse IDE项目。

要在Eclipse IDE中运行这个项目，请单击Package Explorer中的项目名，再单击工具栏中的Run按钮，以启动REPL（可能出现一个对话框，让你选择项目类型，请选择Clojure Application）。启动REPL后，只需在其中执行如下命令即可：

```
(start)
```

切换到Console选项卡，你将看到一条消息，指出服务器已启动。默认不会启动HTTP服务器。要启动它，请切换到REPL选项卡，并执行如下命令：

```
(mount/start #'myapp.core/http-server)
```

现在你可再次访问端口3000处的页面了。要停止这个服务器，请单击Console选项卡中的终止（Terminate）按钮。

8.6.3 探索 Luminus 项目

下表列出了为这个项目生成的最重要的文件和目录。

文件/目录	描　　述
project.clj	与往常一样，project.clj为Leiningen构建文件
profiles.clj	用于存储运行系统所需的数据，如数据库连接凭证。这个文件被配置成不包含在Git版本控制管理器中
src/myapp/config.clj	这是配置文件。它尝试从命令行参数、Java系统属性或子目录env的配置文件中加载设置。可针对如下情形定义不同的设置：`dev`（开发期间使用的设置）、`prod`（用于生产环境的设置）和`test`（运行单元测试时使用的设置）
src/myapp/core.clj	包含`main`方法，即JVM入口函数。它包含启动服务器时调用的函数
src/myapp/layout.clj	包含视图渲染逻辑。默认实现确保模板引擎能够加载子目录/resource中的模板。另外，它还定义了引发异常时加载的错误页面

（续）

文件/目录	描　述
src/myapp/middleware.clj	中间件是包装函数，它们在调用请求处理程序前对请求进行处理。使用模板`myapp`时，生成的一个中间件函数是保护应用程序免受著名Web攻击的中间件函数
src/myapp/handler.clj	这个文件指定有哪些路由可用，以及为每条路由加载哪个中间件。由于URL被关联到路由（参见下一个文件），因此可在多个URL之间共享中间件
src/myapp/routes/home.clj	在这个文件中，定义了URL。对于每个URL，都指定了一个在用户请求它时将调用的处理程序函数。处理程序函数可使用模板引擎来渲染页面。将一组URL关联到一条路由
resources/	这个目录包含静态素材。只有目录resources/public中的文件和子目录可供HTTP服务器使用，其他文件仅供应用程序内部使用
resources/templates/	这个目录包含src/myapp/routes/home.clj使用的HTML模板。对于HTML文件，Luminus默认将模板系统Selmer作为模板引擎
resources/public/	前面说过，这个目录中的一切都可供HTTP服务器使用。所有前端文件都必须存储在这个目录中，这包括图像、JavaScript文件、CSS样式表等

8.6.4　在 Web 应用程序中添加页面

作为练习，下面来给这个应用程序添加一个页面。对这个页面的要求如下。

- 其URL必须为/monadtest。
- 它应使用既有路由`home-routes`。这条路由调用对应用程序进行保护，使其免受特定Web攻击的中间件。
- 它将使用一个HTML页面，这个页面包含一个表单，并使用函数`pretty-msg`来渲染输入的文本。

这个页面将使用本章前面编写的函数`pretty-msg`。

在构建文件project.clj中，添加依赖`org.clojure/algo.monads`：复制文件exploring-monads/project.clj中相应的依赖行，并将其粘贴到文件myapp/project.clj的:dependencies部分。将这个文件存盘后，Counterclockwise将下载这个依赖并将其添加到项目中。

接下来，必须复制项目exploring-monad的文件core.clj，并对其重命名。

- 复制项目exploring-monads中的文件src/exploring-monads/core.clj：在Package Explorer中选择这个文件，再右击它并选择Copy。
- 右击项目myapp的目录src/clj/myapp.routes并选择Paste。
- 右击粘贴的文件src/clj/myapp.routes/core.clj，并选择Refactor>Rename，再输入文件名pretty_msg.clj（注意这里是下划线而不是连字符）并单击OK。

- 打开文件pretty_msg.clj，并将其中的命名空间定义从(ns monad-test.core)改为(ns myapp.routes.pretty-msg)，再将这个文件存盘。

打开文件myapp.routes.home.clj，并在其:require块中添加如下条目：

```
[myapp.routes.pretty-msg :as prettymsg]
```

另外，导入请求方法POST。为此，在:require块中找到下面这一行：

```
[compojure.core :refer [defroutes GET]]
```

再在其中添加方法POST，使其类似于下面这样：

```
[compojure.core :refer [defroutes GET POST]]
```

现在可以编写用户请求/monadtest时将调用的处理程序函数了。它将渲染一个HTML页面，并在该页面中渲染两个变量：prettymsg和msg，其中前者包含经过格式设置后的消息，而后者包含原始消息。为此，在文件myapp.routes.home.clj中，在以(defroutes home-routes打头的代码行前面添加如下代码：

```
(defn monad-test-page [msg]
    (layout/render
        "monadtest.html" {:prettymsg (prettymsg/pretty-msg msg 10)
                          :msg msg }))
```

再在defroutes home-routes块中添加两个条目，为新页面定义URL：

```
(defroutes home-routes
  (GET "/" [] (home-page))
  (GET "/about" [] (about-page))
  (GET "/monadtest" [] (monad-test-page nil))
  (POST "/monadtest" [msg] (monad-test-page msg)))
```

我们将这些条目添加到了home-routes块中，这意味着对URL /monadtest的GET和PUT请求都将使用路由home-routes，而这条路由调用有助于保护应用程序的中间件。请保存对这个文件所做的修改。

最后，添加HTML页面。为此，右击Package Explorer中的目录resources/templates并选择New>Other。在打开的向导中，选择General>File并单击Next按钮，再将文件名设置为monadtest.html。我们首先来定义让用户能够输入句子的表单，为此添加如下代码并将文件存盘：

```
{% extends "base.html" %}
{% block content %}
    <div class="row">
        <div class="col-sm-12">
            <form name="input" action="/monadtest" method="POST">
                {% csrf-field %}
                Message: <input type="text" name="msg" value="{{ msg }}">
                <input type="submit" class="btn" value="Submit">
```

```
            </form>
        </div>
    </div>
{% endblock %}
```

这是一个非常标准的HTML模板。要在生成的输出中替换传递给模板的变量，可使用语法`{{ variable name }}`。在上述代码中，最值得注意的可能是`{% crsf-field %}`。路由`home-routes`调用的中间件可防范**跨站请求伪造**（Cross-Site Request Forgery，CSRF）攻击，这是对网站和应用程序发起的一种常见攻击。使用Luminus提交表单时，必须在HTML中指定一个不可见的`<input>`字段，其中包含该框架生成的一个令牌（token）。`{% csrf-field %}`宏就是负责完成这项任务的。

下面来添加渲染HTML输出的代码，为此在最后一个`</div>`元素后面添加如下代码：

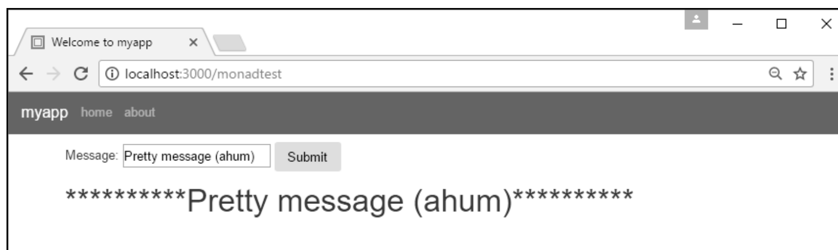
```
<div class="row">
  <div class="col-sm-12">
    <p><h1>{{ prettymsg }}</h1></p>
  </div>
</div>
```

确保当前没有运行任何`myapp`实例；如果必要，在Console选项卡中终止当前会话，并单击REPL选项卡的关闭图标将其关闭。然后，运行项目myapp：在Package Explorer中单击这个项目，并单击工具栏中的Run按钮，再选择Clojure Application，然后在REPL启动后输入`(start)`并按回车。重新切换到REPL选项卡，再执行命令`(mount/start #'myapp.core/http-server)`。

在你喜欢的浏览器中，访问下面的新URL：

http://localhost:3000/monadtest

如果一切顺利，你将看到一个页面，可在其中输入并提交消息：



8.7　小结

本章首先在Eclipse IDE中安装了插件Counterclockwise，其功能虽然没有本章介绍的其他Eclipse插件那么强大，但也很好使。我们还安装了Leiningen——最受Clojure开发人员欢迎的构建

工具。遵守前一章的承诺，我们还介绍了如何在使用和不使用构建工具Leiningen的情况下编译类文件；还尝试使用了便利的Leiningen任务uberjar，它生成一个包含所有依赖项的JAR文件。我们创建了第一个项目，并使用单元测试框架`clojure.test`以测试驱动开发的方式探索了monad。接下来，使用微框架Luminus基于内置模板创建了一个Web项目，并将其导入到Eclipse IDE中。我们在这个项目中添加了一个页面，它接受文本输入，并使用8.5节编写的函数显示这些文本。

本书接下来要介绍的语言是Kotlin。与Java一样，Kotlin也是一种静态的强类型编程语言，但相比于使用Java，使用Kotlin编写的代码要紧凑得多，同时也很容易理解。

第 9 章 Kotlin

Kotlin是JetBrains公司设计的一款语言，这家公司推出了众多流行的IDE，支持使用各种语言进行编程，包括Java（IntelliJ）、Python（PyCharm）、PHP（PhpStorm）等。这些IDE有商业版，也有免费的社区版（社区版的功能更少，但也很有用）。与Java一样，Kotlin也是一种静态类型语言，主要是为面向对象编程而设计的，但也支持过程性编程。与众多现代OOP语言一样，它也提供了很多函数式编程功能。本章介绍如下主题：

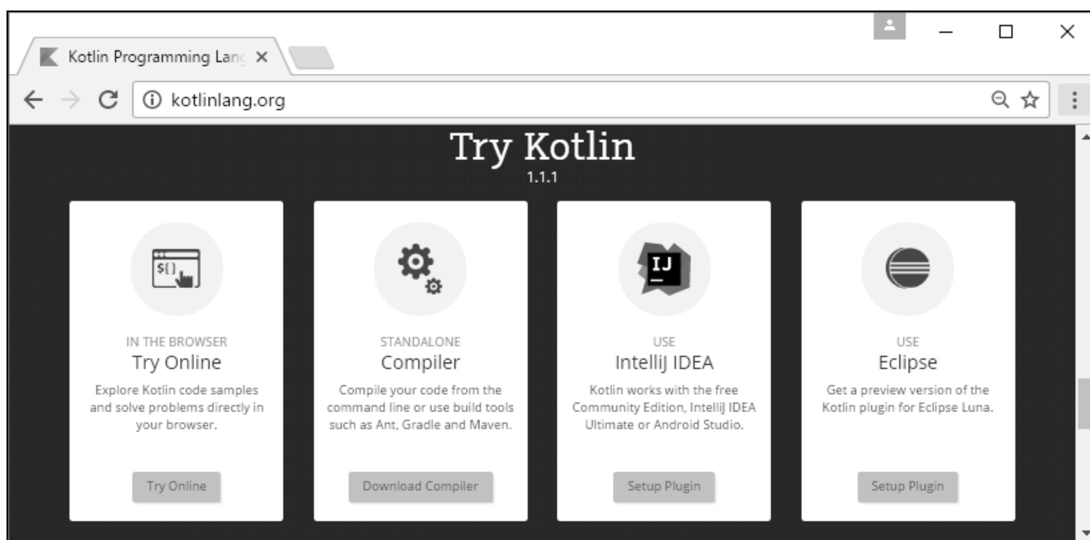
- 安装Kotlin；
- Kotlin的REPL交互式shell；
- Kotlin语言基础；
- Kotlin的OOP功能；
- 使用Kotlin进行过程性编程；
- 风格指南；
- 小测验。

9.1 安装 Kotlin

要下载Kotlin编译器，可访问Kotlin官网。下载或运行编译器的方式有多种：

- 在在线版Kotlin中运行代码片段；
- 下载编译器。

为尝试运行本章的示例，最好是下载独立版编译器。编写本书期间，可从GitHub下载独立版编译器，下载地址可在http://kotlinlang.org找到。



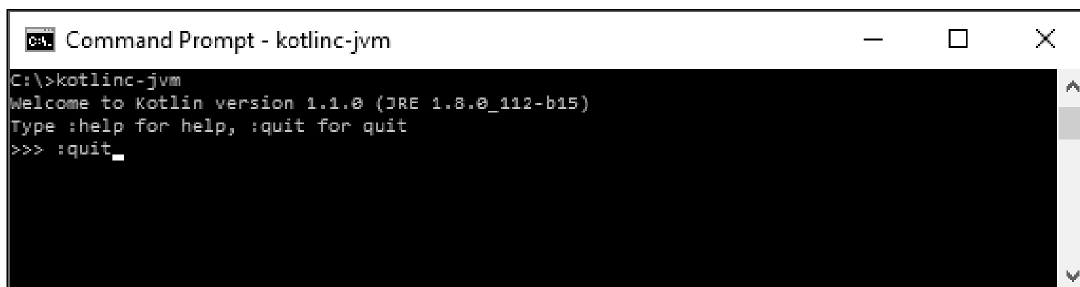
要下载最新的独立版编译器，请按如下说明操作。

(1) 在Kotlin官网主页中向下滚动，找到方框STANDALONE Compiler，并单击其中的链接Download Compiler。这将显示一篇文章，其中介绍了如何下载最新的独立版编译器，还提供了到GitHub页面的链接。
(2) 在GitHub仓库网站，向下滚动到Downloads部分，其中有包含最新版的ZIP文件。下载这个ZIP文件。本书出版时，最新版为1.1，ZIP文件名为kotlin-compiler-1.1.zip。

Kotlin的安装过程与本书介绍的其他语言很像。

(1) 将文件解压缩。
(2) 将解压缩得到的目录bin添加到环境变量Path中。

要检查安装情况，可尝试运行启动脚本kotlinc-jvm（在Windows中为kotlinc-jvm.bat；在Linux和macOS中为kotlinc-jvm）。这将启动交互式shell（也被称为REPL），就像前几章中一样。



要退出这个shell，可输入:`quit`并按回车。



不同REPL shell的设计者并未就该提供哪些命令达成一致。正如你在前几章看到的，Scala和Clojure（使用Leiningen启动shell时）使用:`exit`来退出shell，而Kotlin REPL使用:`quit`。

启动脚本

Kotlin自带了多个用于不同操作系统的启动脚本。鉴于Kotlin编译器能够编译到多个目标（JVM和JavaScript），因此针对每个目标提供了不同的启动脚本。当前，默认目标为JVM，因此你也可使用通用启动脚本kotlinc来编译Kotlin代码。下面概述了目录bin中的启动脚本。

Windows启动脚本	Linux/macOS启动脚本	描　述
kotlinc.bat	kotlinc	启动默认的Kotlin编译器实现（默认目标为JVM）
kotlinc-jvm.bat	kotlinc-jvm	启动将Kotlin代码编译为JVM字节码的Kotlin编译器；在无需指定命令行选项时，也可用于启动REPL
kotlinc-js.bat	kotlinc-js	启动将Kotlin代码编译为JavaScript代码的Kotlin编译器，这种代码可用于Web应用程序的前端中。这个编译器没有REPL
kotlin.bat	kotlin	这个脚本可用于运行Kotlin编译器编译得到的类文件中的`main()`函数，它自动将Kotlin运行时库添加到Java类路径中

鉴于本书的重点是JVM，因此这里不再涉及编译为JavaScript代码的情形。



Kotlin现已成为Android平台的官方语言，这意味着Google已将Kotlin视为一级Android软件开发语言。较新的Android Studio IDE都自带Kotlin。第1章说过，虽然Android使用Java，但本书不涉及这个方面。

9.2　Kotlin 的 REPL 交互式 shell

与本书前面介绍的两种语言（Scala和Clojure）一样，Kotlin也提供了REPL交互式shell，可用于以交互的方式尝试运行Kotlin代码片段。前一节说过，要启动这个REPL，可在不指定任何参数的情况下运行编译器启动脚本（在Windows中，可加上扩展名.bat，但并非必须这样做）：

```
kotlinc-jvm
```



你也可以运行启动脚本kotlinc，因为JVM是Kotlin的默认编译目标。

REPL shell实现了几个内置命令。在Kotlin REPL中，无需调用Java类库方法来退出shell。

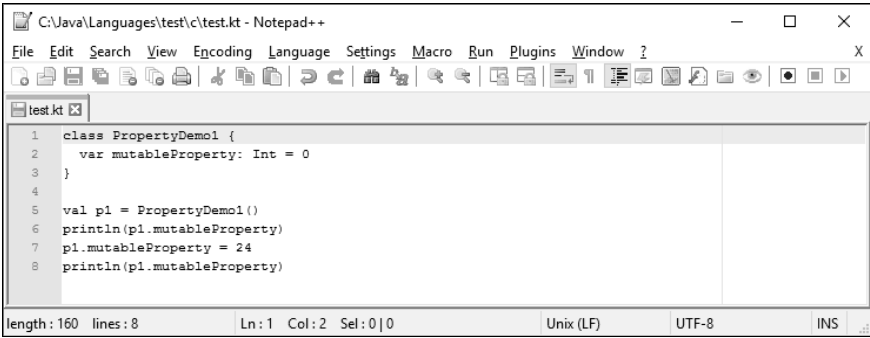
命　　令	描　　述
:help	显示包含REPL内置命令的帮助屏幕
:quit	用于退出REPL
:dump bytecode	对于当前会话期间生成的所有代码，将其转换为Java字节码并以可读的文本格式存储。对大多数最终用户来说，这没什么用，但想研究Java字节码的高级程序员会感兴趣
:load FILE	在当前REPL实例中加载文件（请将FILE替换为包含Kotlin源代码的文件的完整路径）

编写本书期间，Kotlin REPL好像存在一些严重的可靠性问题，尤其是在Windows平台中上。复制并粘贴代码时，它常常拒绝使用独立编译器能够编译的代码；另外，编译完全正确的代码时，这个程序时不时会挂起。但愿这些问题在以后的版本中能够得到解决。本章的代码在REPL中都能运行。

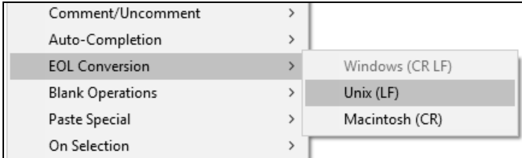


如果你在REPL中输入代码时遇到问题，可尝试将这些代码放在一个扩展名为kt的源代码文件中，再使用REPL的命令:load来加载并处理这个文件。需要注意的是，如果你使用的是Windows，必须将源代码文件转换为Linux换行（end of lines，EOL）格式。

在Windows系统中使用命令:load时，必须使用能够将文件保存为Linux EOL格式的文本编辑器，因为编写本书期间，命令:load不支持Windows的CR + LF换行格式。一个这样的免费开源编辑器是Notepad++，该编辑器可从http://notepad-plus-plus.org下载。



在Notepad++将文件存盘前，请选择菜单Edit>EOL Conversion>Unix (LF)：



Packt网站提供的示例文件都是以Linux EOL格式存储的，以便你更轻松地运行它们。

9.3　Kotlin 语言基础

学习Kotlin时，建议你随时参阅Kotlin参考文档。要找到这个文档，可单击Kotlin官网主页中的链接LEARN，也可直接访问http://kotlinlang.org/docs/reference/。

本节介绍如下主题：

- 定义局部变量；
- 定义函数；
- Kotlin类型；
- 循环。

9.3.1　定义局部变量

要定义局部变量，可使用`var`或`val`：

```
var aMutableNumber = 24
val anImmutableNumber = 42
```

区别在于使用`var`定义的变量是可以修改的，而使用`val`定义的变量是不可修改的。定义变量时还可指定其类型：

```
var aMutableString: String = "A type can optionally be specified..."
val anImmutableString: String = "...no matter whether you are using
                                 var or val"
```

支持的类型将在9.3.3节讨论。要将`null`赋给变量，务必采取必要的预防措施。Kotlin的类型系统比较独特，针对可为`null`的变量制订了一些使用规则，这也将在9.3.3节讨论。

变量可在函数或类中定义，也可在代码开头定义（采用过程性编程时），这些情形都将在相关的章节中讨论。

你可按下面的规则使用字面量：

- 不带后缀的整数为`Int`；
- 要使用`Long`字面量，必须指定后缀L；
- 可使用十六进制表示`Int`和`Long`，为此可加上前缀0x；
- 可使用二进制表示整数，为此可加上前缀0b；
- 不带后缀的浮点数为`Double`；
- 带后缀f或F的浮点数为`Float`；
- `Float`和`Double`值都可使用科学记数法表示。

下面是一些示例：

```
val thisIsAnInt = 42
val thisIsALong = 1000L
val hexInt = 0xFF
val binaryLong = 0b10101100L
val thisIsADouble = 149.16
val thisIsAFloat = 501.19e2f
```

Kotlin不像Java那样使用关键字new来实例化对象，而像使用函数那样使用类名：

```
class A (i: Int) {
}
val a = A(25)
```

上述代码定义了类A，它包含接受一个Int输入参数的主构造函数。接下来，实例化了这个类：在类名后加上构造函数参数的值。

9.3.2 定义函数

要定义函数，可使用关键字fun：

```
 fun functionName() {
}
```

当然，也可给函数指定参数，但指定参数时必须指定其类型：

```
fun functionNameWithParameters(i: Int, j: Int) {
  println(i * j)
}
```

如果没有指定返回类型（就像前面两个示例那样），函数的返回类型默认为Unit。Unit相当于Java中的void，不同之处在于可将Unit返回值赋给变量：

```
fun noReturnValue(x: Int, y: Int): Unit {
val f = noReturnValue(1, 2)
println(f)
```

如果在REPL中输入并执行上述代码片段，将打印kotlin.Unit。下面是一个返回整数（Kotlin类型Int）的函数：

```
fun returnsAnInt(x: Int, y: Int): Int {
  return x * y
}
val f = returnsAnInt(10, 10)
println(f)
```

如果函数只包含一行代码，定义它时可不使用大括号（{ }），而使用运算符=。对于只包含一行代码的函数，可不显式地指定返回类型，因为返回类型可根据代码推断出来：

```
fun alsoReturnsAnInt(x: Int, y: Int) = x * y
```

9.3.3　Kotlin 类型

Kotlin的一个独特之处是其类型系统，它在处理null引用方面尤其独特。本节讨论如下主题：

- Kotlin基本类型；
- 字符串；
- 安全地处理null；
- 转换；
- 集合和泛型。

1. Kotlin基本类型

Kotlin不直接使用JVM数据类型，而是将它们包装成自己的类型，这样做的原因之一是Kotlin支持多个编译目标（当前为JVM、Android和JavaScript）。通过使用自己的类型，可确保不同平台上的功能一致。但Kotlin依然与使用JVM数据类型（如基本类型以及java.lang.Integer和java.lang.String等常见类）的JVM代码完全兼容，这是因为调用Java代码时，Kotlin编译器会自动在JVM类型和Kotlin内部类型之间进行转换。

下表列出了Kotlin中最重要的基本类型及其全限定类型名，以及对应的JVM类型。

Kotlin类型名	Kotlin全限定类型名	对应的JVM类型
Byte	kotlin.Byte	基本类型byte
Byte?	kotlin.Byte?	java.lang.Byte
Double	kotlin.Double	基本类型double
Double?	kotlin.Double?	java.lang.Double
Float	kotlin.Float	基本类型float
Float?	kotlin.Float?	java.lang.Float
Int	kotlin.Int	基本类型int
Int?	kotlin.Int?	java.lang.Integer
Long	kotlin.Long	基本类型long
Long?	kotlin.Long?	java.lang.Long
Short	kotlin.Short	基本类型short
Short?	kotlin.Short?	java.lang.Short
Any	kotlin.Any	java.lang.Object
String	kotlin.String	java.lang.String

稍后将详细说明类型名后面的问号的含义。就目前而言，你只需知道仅当变量是带问号的类型时，才能为null引用就够了。其类型不带问号的变量不能为null。

有趣的是，Kotlin类型很像普通类（例如，每种类型好像都有方法），但在内部，前述很多类型都在可能的情况下直接使用JVM基本类型。这方面的工作完全是由编译器在幕后处理的，这极大地改善了性能，因为无需不分青红皂白地将基本类型值自动装箱。

2. 字符串

类型kotlin.String功能强大却易于使用，你可以像使用Java类java.lang.String那样使用它：

```
val s: String = "Hello!"
```

Kotlin也支持原始字符串，这种字符串可横跨多行：

```
val s: String = """
raw string"""
```

不同于常规字符串，在原始字符串中，可使用斜杠对字符进行转义（例如，\n表示换行符，\t表示制表符）。

还支持字符串模板：

```
  var favoriteBar = "FooBar"
println("Your favorite bar's name $favoriteBar consists of
      ${favoriteBar.length} characters")
```

这将打印Your favorite bar's name FooBar consists of 6 characters。



请注意，原始字符串不支持字符串模板。

3. 安全地处理null

前面多次说过，Kotlin可避免将null赋给引用变量导致的错误。对于普通类型变量，如果你将null赋给它，Kotlin将拒绝编译。

例如，下面的代码无法通过编译：

```
var currentTime = java.util.Date()
// 下面这行代码无法通过编译
currentTime = null
```

前面说过，实例化类时，Kotlin不要求你使用关键字new（Kotlin根本就不支持这个关键字）。如果你运行上述代码，将出现如下错误：

```
error: null can not be a value of a non-null type Date
```

要让这些代码通过编译，必须在变量的类型名后面加上问号：

```
var currentTime: java.util.Date? = java.util.Date()
// 现在这行代码能够通过编译
currentTime = null
```

要调用currentTime的方法或访问其其他成员，必须让编译器知道，这个引用可能为null，

也可能不为null。例如，下面的代码无法通过编译：

```
var currentTime: java.util.Date? = java.util.Date()
// 下面这行代码无法通过编译
var seconds = currentTime.getTime()
```

这将导致如下错误：

```
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a
nullable receiver of type Date?
```

在上述代码片段中，虽然currentTime不是null引用，但你必须告诉Kotlin编译器，你知道currentTime有可能为null。解决方案有多种：

- 添加条件检查；
- 使用安全调用运算符?.；
- 使用Elvis运算符?:；
- 使用运算符!!。

- *方法1：添加条件检查*

通过添加一条if语句，可告诉编译器，你知道引用变量有可能为null：

```
fun test() {
  var currentTime: java.util.Date? = java.util.Date()
  println("Line below will now compile fine")
  var seconds = if (currentTime != null) currentTime.getTime() else 0
  println(seconds)
}
test()
```

编译器得知你意识到这个实例变量可能为null后，就会心满意足地编译代码。以这种方式使用时，if条件返回currentTime.getTime()或0。在前述示例中，调用test()时将打印currentTime.getTime()的输出，因为currentTime不是null引用。

请注意，仅当编译器知道其他线程无法访问这个变量时，这种做法才管用。由于currentTime是在函数中定义的，其他线程确实无法访问它。如果currentTime是一个类的公有字段，编译器依然会拒绝编译这些代码，因为在执行if(currentTime != null)检查之后到执行currentTime.getTime()调用之前，其他线程可能修改currentTime。在这种情况下，必须采用其他方法，否则编译器将拒绝编译，并显示一条错误消息。

- *方法2：使用安全调用运算符?.*

Kotlint提供了安全调用运算符?.（问号后面跟一个句点），它在引用为null时返回null，否则就调用指定的方法或访问指定的成员：

```
var currentTime: java.util.Date? = null
var seconds = currentTime?.getTime()
println(seconds)
```

调用test()时，将打印null，因为执行currentTime?.getTime()时，Kotlin发现currentTime是一个null引用。如果currentTime指向一个java.util.Date实例，Kotlin将打印方法getTime()的输出。

运算符?.的一个优点是可以串接，下面是一个虚构的示例：

```
member1?.member2()?.member3()
```

如果属性member1为null引用或方法member2返回null，整个表达式的结果都将为null。如果member1和member2的输出都不是null引用，将返回方法member3的输出。

- 方法3：使用Elvis运算符?:

对于第一个示例中的if语句，可使用Elvis运算符?:进行改写，得到的代码更简洁：

```
var currentTime: java.util.Date? = null
var seconds = currentTime?.getTime() ?: -1
println(seconds)
```

上述代码将返回-1。这是因为currentTime?.getTime()返回null（安全调用运算符?.返回null，因为currentTime为null引用），因此返回Elvis运算符?:后面的字面量-1。如果currentTime指向一个java.util.Date实例，上述代码将打印getTime()的输出。

- 方法4：使用!!运算符

Kotlin官方文档指出，这个运算符就是为喜欢异常NullPointerException的人设计的。

通过在变量名后面加上运算符!!，可让Kotlin编译器完全忽略null安全系统。如果这个实例变量为null引用，而代码试图调用其方法或访问其成员，Kotlin将引发NullPointerException异常，就像Java在这种情况下的做法一样：

```
 fun test() {
   var currentTime: java.util.Date? = null
  println("Next line compiles, but throws exception when running")
var seconds = currentTime!!.getTime()
  println(seconds)
 }

test()
```

4. 转换

在Java中，编译器在确定不会降低精度时，将自动进行转换。例如，下面的做法在Java中是合法的：

```
// Java代码
int a = 1000;
long b = a;
```

由于将int值存储到long变量中时，不会降低精度，因此Java自动将其转换为long。Kotlin不会自动转换变量，而要求程序员手动进行转换：

```
val a: Int = 1000
val b: Long = a.toLong()
```

为支持转换，Kotlin的每种数值类型（Int、Long等）都包含如下方法：

- toByte()；
- toChar()；
- toDouble()；
- toFloat()；
- toInt()；
- toLong()；
- toShort()。

5. 集合和泛型

与Scala和Clojure一样，Kotlin也提供了自己的集合类实现，包括常见集合类的可修改和不可修改版本。Kotlin的泛型实现与Java很像。下表列出了Kotlin支持的接口，以及创建Kotlin运行时库中实现了这些了接口的类的实例时，必须调用的函数：

接　　口	描　　述	用于创建实例的函数
List<T>	为不可变列表提供方法	listOf
MutableList<T>	为可变列表提供方法	mutableListOf
Set<T>	为不可变集提供方法	setOf
MutableSet<T>	为可变集提供方法	mutableSetOf
Map<K, V>	为不可变映射提供方法	mapOf
MutableMap<K, V>	为可变映射提供方法	mutableMapOf

有关每种类型包含的各种方法和属性的详尽说明，请参阅Kotlin文档的API参考（API Reference）部分。下面通过示例说明上表列出的一些类型的方法，先看来一个不可变列表：

```
val someImmutableInts: List<Int> = listOf(10, 20, 3
println("$someImmutableInts --> ${someImmutableInts.size} elements")
```

这个代码片段打印[10, 20, 30] --> 3 elements。Kotlin接口List的其他方法包括contains()、indexOf()、isEmpty()、lastIndexOf()和subList()。接口List还包含各种函数式编程函数，这些函数被称为扩展函数。扩展函数将在下一章介绍。

```
 val mutableDoubles: MutableList<Double> = mutableListOf(3.14, 1.0,
                                                          25.5)
 mutableDoubles.add(1, -1.99)
 mutableDoubles.removeAt(0)
println(mutableDoubles)
```

上述代码打印[-1.99, 1.0, 25.5]。接口MutableList的其他常用函数包括addAll()、clear()和remove()，它们用于删除特定的元素。

```
val mapNumbers: Map<String, Int> = mapOf("one" to 1, "ten" to 10,
                                          "thirty" to 30)
println(mapNumbers["thirty"])
for ((key, value) in mapNumbers) {
  print("$key = $value ")
}

println()
```

这个示例打印30和one = 1 ten = 10 thirty = 30。Map的其他常用方法包括keys()、values()、containsKey()、containsValue()、getOrDefault()（1.1版新增的）和isEmpty()。

9.3.4 循环

Kotlin支持所有常见的循环语句，如for、while和do...while。

首先来看一个for循环示例：

```
val items = listOf(10, 20, 30)
for (i in items) {
  println(i)
}
```

一点都不出乎意料。这段代码打印10、20和30，每个元素各占一行。

还有while语句。与其他语言中一样，while语句先检查条件，如果为false，就不做任何迭代，否则就开始循环，直到条件为false或调用了方法break：

```
var x = 10
while (x > 20) {
  println("Hello")
  x++
}
```

这个示例什么都不打印，因为x不比20大。

另外，还有变种do...while：

```
var y = 0
do {
  y++
```

```
  if (y == 2)
    continue
  println(y)
} while (y % 5 != 0)
```

这段代码打印1、3、4、5。

与Java和众多其他流行的编程语言一样，所有Kotlin循环结构都支持break和continue语句，它们分别停止迭代和跳过当前迭代。

9.4 Kotlin 的 OOP 功能

Kotlin首先是一种OOP语言，本节将全面介绍Kotlin的OOP基本知识：

- 定义包；
- 导入成员；
- 定义类和构造函数；
- 给类添加成员；
- 继承；
- 可见性限定符；
- 单例对象和伴生对象；
- 数据类；
- lambda和内联函数。

9.4.1 定义包

包是使用package语句定义的，这种语句的工作原理与Java中很像：

```
package com.example
```

不同于Java和Clojure，在Kotlin中，源代码的目录结构无需与包名匹配；换言之，你可以随心所欲地组织源代码。



请不要在Kotlin交互式REPL中使用package语句，因为它不支持创建包。

9.4.2 导入成员

Kotlin import语句与Java import语句很像：

```
import java.util.ArrayList
import java.io.*
```

一个重要的不同是可指定别名，这在可能发生名称冲突的情况下提供了极大的便利：

```
import java.io.File as JavaFile
val f = JavaFile("test.txt")
```

9.4.3 定义类和构造函数

类是使用关键字class定义的：

```
class ClassName {
}
```

可在类头中指定主构造函数：

```
class Point constructor(x: Int, y: Int) {
}
```

关键字constructor可以省略：

```
class Point (x: Int, y: Int) {
}
```

如果要在实例化类时执行一些代码，可使用关键字init指定一个代码块：

```
class Point (x: Int, y: Int) {
  init {
    println("Executable code here...")
  }
}
```

在init代码块中，可使用构造函数的参数以及当前类中定义的属性（至于如何添加属性，稍后将介绍）。对于要在方法中使用的构造函数参数，必须给它们加上前缀val（用于不可变属性）或var（用于可变属性）：

```
class Point (val x: Int, val y: Int) {
  override fun toString(): String { return "${x}, ${y}" }
}
val p = Point(-30, 50)
println(p)
```

这将打印-30, 50。通常，最好使用val将构造函数参数设置为不可变的，因此除非出于设计考虑，否则在使用var将构造函数设置为可变的之前一定要三思而后行。

指定了关键字constructor时，还可给主构造函数指定访问限定符，但在省略了关键字constructor的情况下，无法这样做（在这种情况下，构造函数默认为公有的）：

```
class Customer private constructor(id: Int) { }
```

在没有显式地定义构造函数的情况下，将自动生成一个默认构造函数，这个构造函数是公有

的且不接受任何参数。如果你不希望自动创建这样的构造函数，可显式地定义一个不接受任何参数的私有构造函数，如下所示：

```
class Customer private constructor()
```

可使用的访问限定符将在后面介绍。还可添加一个或多个辅助构造函数：

```
class Customer(val name: String, val country: String?) {
  constructor(name: String) : this(name, null) {
      println("Name: " + name)
      println("Country: " + country)
  }
}

var c = Customer("Your Name")
```

上述代码片段将打印`Name: Your name`和`Country: null`。辅助构造函数必须直接（如上例所示）或间接地调用主构造函数。所谓间接地调用主构造函数，指的是调用另一个辅助构造函数，而这个辅助构造函数直接或间接地调用了主构造函数。

9.4.4　给类添加成员

下面来介绍如何给类添加成员：

- 添加函数；
- 添加入口函数`main`；
- 添加属性。

1. 添加函数

在Java中，类中的函数被称为方法，但在Kotlin中，它们也被称为函数。由于函数在前面介绍过，因此添加函数的方式完全在意料之中：

```
class MethodDemo {
  fun instanceMethod(i: Int): Int {
      return i*i
  }

var demo = MethodDemo()
println(demo.instanceMethod(5))
```

有趣的是，Kotlin没有定义静态方法（类方法）的关键字。作为一种替代办法，你可将函数放在类外面（这个主题将在9.5节讨论）。你也可以使用伴生对象来生成静态函数，这也将在本章后面讨论。

- 入口函数`main`

你知道，在Java中，可添加`static main(String[] args)`，它将作为应用程序的入口。

在Kotlin中，类中的方法自动为实例方法，因此在类中添加如下函数不管用：

```
// 下面的main()函数是普通的实例方法
// 不能作为应用程序的入口
class A {
  fun main(args : Array<String>) {
    println("Executable code here...")
  }
}
```

在Kotlin中，定义入口`main()`的方法有两种。

- 将函数`main()`放在源代码的最顶层（不在任何类中），这将在9.5节讨论。
- 将函数`main()`放在一个伴生对象中，并添加`@JvmStatic`注解，这将在9.4.8节演示。

2. 添加属性

在Kotlin中，类不能包含独立的变量，但可包含属性。属性是有配套获取函数和/或设置函数的变量（具体情况取决于属性是否是可读和/或可写的）Kotlin可为属性生成默认的获取/设置函数，但你也可提供获取/设置函数的实现。

下面是一个可变（可读写）的属性；这个属性之所以是可变的，是因为声明它时使用了关键字`var`：

```
class PropertyDemo1 {
  var mutableProperty: Int = 0
}

val p1 = PropertyDemo1()
println(p1.mutableProperty)
p1.mutableProperty = 24
println(p1.mutableProperty)
```

属性`mutableProperty`被初始化为0。由于没有显式地提供获取函数和设置函数的实现，Kotlin编译器将自动为这个属性生成获取函数和设置函数。要访问这个属性，只需使用属性名，而无需添加前缀get或set。Kotlin自动调用生成的获取函数和设置函数，这些函数在Kotlin中被称为存储函数（accessor）。



Kotlin编译器要求要么在声明属性的同时对其进行初始化（如前面的示例所示），要么在`init { }`块中对其进行初始化。

下面是一个只读的属性；只读属性是使用关键字`val`声明的：

```
class PropertyDemo2 {
  val readOnlyProperty: Int = 1000
}
```

9

```
val p2 = PropertyDemo2()
println(p2.readOnlyProperty)
```

如果你试图执行类似于前一个示例的代码p2.readOnlyProperty = 1234，将引发类似于下面的异常：java.lang.IllegalAccessError: tried to access field Line35$PropertyDemo2.readOnlyProperty from class Line39。像可变属性一样，对于只读属性，也必须在声明时显式地初始化或在init { }块中初始化。

前面说过，可显式地给属性定义存取函数（获取函数和设置函数），下面就是一个这样的可变属性：

```
class PropertyDemo3 {
  var customProperty: Int = 1000
  get() { field + 1 }
  set(value) { field = value }
p3.customProperty = 10
println(p3.customProperty)
```

关键字field用于访问生成的字段，Kotlin文档称之为**支持字段**（backing field）。

可将设置函数设置为私有的以隐藏它：

```
class PropertyDemo4 {
  var anotherProperty: Int = 314
    private set
}

var p4 = PropertyDemo4()
println(p4.anotherProperty)
```

属性anotherProperty确实有设置函数（由Kotlin提供的默认实现），但由于它是私有的，因此被隐藏了。但与往常一样，获取函数被自动创建且是公有的。然而，你不能将获取函数设置为私有的；获取函数必须与属性的访问限定符匹配。在Kotlin中，访问限定符被称为可见性限定符，将在后面更详细地讨论。

9.4.5　继承

与大多数流行的JVM语言一样，Kotlin也只允许类最多扩展一个超类。对于没有显式地继承其他类的类，将继承Kotlin类kotlin.Any。kotlin.Any类似于Java类java.lang.Object，但需要注意的是，它们是两个不同的类。

创建时没有显式指定限定符的类都是final的，不能继承。要让类能够被继承，必须在定义它时在前面加上关键字open：

```
open class Person(name: String)
class Customer(name: String, department: String) : Person(name) {
}
```

如果子类没有主构造函数，要调用父类的构造函数，必须使用关键字`super`：

```
open class Person(name: String)
class Customer : Person {
  constructor(name: String) : super(name)
}
```

要继承方法，必须使用关键字`override`：

```
open class ParentClass {
  open fun greatMethod() {
    println("greatMethod in parent class")
  }
}

class ChildClass: ParentClass() {
  override fun greatMethod() {
    super.greatMethod()
  }
}
```

请注意，除非显式地使用了访问限定符`open`，否则函数默认为`final`的。

9.4.6 接口

Kotlin的接口类似于Java 8中的接口，可包含抽象函数（没有实现的函数）和具体函数（带默认实现的函数）：

```
interface NameOfInterface {
  fun functionWithoutImplementation()
  fun functionWithImplementation(i: Int) {
    // 下面是默认实现……
  }
}
```

在Kotlin接口中，还可声明属性（这些属性可以有显式的获取函数，也可以没有）：

```
interface InterfaceWithProperties {
  var propertyWithGetterAndSetter: Int
  val propertyWithGetterOnly: String
  val propertyWithDefaultImplementation: Double
    get() = 0.0
}
```


接口不可能包含支持字段，因此在接口中不能使用关键字`field`。有鉴于此，对于接口中的属性，无法给它显式地提供设置函数实现。

Kotlin使用相同的语法来实现接口和继承类：

```
class DemoClass : NameOfInterface, InterfaceWithProperties {
  override fun functionWithoutImplementation() {
    println("but now it has a implementation")
```

```
  }
  override var propertyWithGetterAndSetter: Int = 0
  override val propertyWithGetterOnly: String = "test"
}
```

类和接口的排列顺序无关紧要，但如果有超类，最好将它放在最前面，然后再指定类实现的接口。

9.4.7　可见性限定符

在Kotlin中，可在类外定义函数和属性；类外被称为包的顶层（top level），将在9.5节更详细地介绍。对于顶层声明和类成员，可使用的可见性限定符（在Java中被称为访问限定符）不同。下表列出了Kotlin支持的可见性限定符。

可见性限定符	可　用　于	描　　述
`public`	顶层声明和类成员	这是在Kotlin中没有显式指定可见性限定符时默认使用的可见性限定符，其函数与Java访问限定符public相同：声明可在任何地方使用
`private`	顶层声明和类成员	相应的定义仅对当前文件中的代码可见
`internal`	顶层声明和类成员	相应的定义仅对当前模块中的代码可见。模块可以是一组一起编译的Kotlin文件，如特定项目的所有Kotlin源代码文件
`protected`	仅限于类成员	与Java中相同：成员仅对当前类及其子类可见，对其他类都不可见

在Java中，没有指定访问限定符时，成员默认是包私有的，但Kotlin没有与之对应的可见性限定符。

9.4.8　单例对象和伴生对象

Kotlin关键字`object`与Scala关键字`object`很像，也用于创建单例：

```
object ThisIsASingleton {
  fun coolMethod() = println("Not so cool, after all")
}

ThisIsASingleton.coolMethod()
```

将自动创建这个类的一个实例，并可通过对象名来访问其成员。

可在类中创建单例对象，为此需要添加前缀`companion`：

```
class NormalClass {
  companion object CompanionObject {
    var i = 100
    fun yetAnotherCoolMethod() {
      i = 50
    }
  }
```

```
}

NormalClass.CompanionObject.yetAnotherCoolMethod()
println(NormalClass.i)
println(NormalClass.CompanionObject.i)
```

正如这里演示的，要访问伴生对象的成员，可通过引用NormalClass.Companion Object，也可只使用类名NormalClass（因为它也指向伴生对象CompanionObject）。上述代码片段将打印50两次。



如果没有显式地指定伴生对象的名称，访问其成员时可指定伴生对象。在这种情况下，Kotlin将伴生对象称为Companion。

要访问伴生对象，只能通过其所属的类（这里为NormalClass），而不能通过引用变量来访问。因此，下面的代码无法通过编译：

```
var i = Normalclass()
// 无法通过编译
i.CompanionObject.yetAnotherCoolMethod()
```

这些代码将引发如下异常：error: nested companion object 'CompanionObject' accessed via instance reference。

之所以会出现这种错误，是因为伴生对象很像Java中的静态成员。由于伴生对象是单例对象，因此它只有一个实例，由其所属类的所有实例共享。为让你知道你使用的不是普通的实例变量字段和函数，Kotlin禁止通过引用变量访问伴生对象。

需要指出的是，严格地说，伴生对象及其成员并不是静态成员。在内部，它们依然是普通的实例成员，但由于在程序开始使用前就被自动实例化，因此它们很像静态成员。在伴生对象中，可生成真正的静态方法，为此可使用注解@JvmStatic：

```
class StaticDemo {
  companion object {
    @JvmStatic fun realStaticMethod() {
      println("Real static method...")
    }
  }
}

StaticDemo.realStaticMethod()
```

同样，要调用这个方法，只需通过伴生对象所属类的名称即可。当你这样做时，Kotlin将确保调用的是伴生对象的静态方法realStaticMethod()。

要将对象或伴生对象中的字段编译成真正的JVM静态字段，可在它前面添加关键字const：

```
class StaticFieldsDemo {
  companion object {
```

```
    const val CONSTANT_VALUE = 3
  }
}
```

这与Java代码`public static final int CONSTANT_VALUE = 3;`等效。



你可能会问，考虑到伴生对象的行为很像静态成员，为何要创建真正的静态方法或静态字段呢？原因之一是对被其他JVM语言（如Java）调用的类来说，这可能很有用。

一个典型的真正的静态方法是应用程序的JVM入口方法`main()`。根据设计，这个方法必须是静态的。在Kotlin中，要在类中定义入口，唯一的办法是创建一个伴生对象并使用注解`@JvmStatic`：

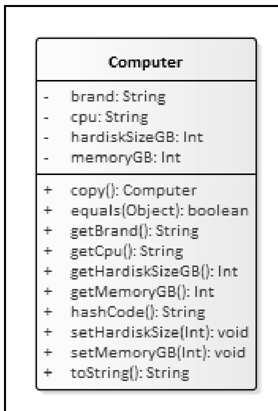
```
class MainDemo {
  companion object {
    @JvmStatic fun main(args: Array<String>) {
      println("This is the main method")
    }
  }
}
```

9.4.9　数据类

要创建只包含几个字段的类，数据类将很有用。第3章说过，在Java中创建POJO类时，必须编写大量的代码来定义字段并为每个字段定义两个方法（获取函数和设置函数）。Kotlin提供了数据类，它自动为每个字段生成属性。来看一个示例：

```
    data class Computer(val brand: String, val cpu: String, var memoryGB:
Int, var harddiskSizeGB: Int)
```

这行代码生成的类类似于下面这样：



在代码中，访问数据类的方式与访问Kotlin普通类相同；另外，其主构造函数将所有字段作为参数：

```
var pc = Computer("Dell", "Intel Core i5", 8, 1024)
println(pc.brand)
pc.memoryGB = 4
```

对于数据类，编译器自动为每个字段创建一个获取函数，对于使用关键字var而不是val定义的每个可变字段，还自动为其创建一个设置函数。编译器还自动给数据类添加常用JVM方法（如equals()、hashCode()和toString()）的实现。

在Kotlin中，访问属性时，无需添加前缀get/set，前面的代码片段演示了这一点。

编译器还自动为数据类生成函数copy()，这提供了极大的便利，让你能够创建数据类实例的副本，并指定要在副本中修改哪些字段：

```
var pc2 = pc.copy(brand="HP", memoryGB=16)
println(pc2)
```

这将创建变量pc指向的数据类Computer的实例的副本，但将字段brand（品牌）改成了HP，将字段memory（内存量）提高到了16GB。



数据类可实现接口，在Kotlin 1.1版中还可扩展类。

9.4.10 lambda 和内联函数

与本书介绍的其他大多数语言一样，在Kotlin中，也可将lambda函数作为参数进行传递。假设有一个应用程序有足够的权限，能够重启服务器。那么调用重启服务器的函数时，你可能想将这种信息写入到某个地方。这个shutdown函数的函数头如下：

```
fun shutdown(logger: (m: String) -> Unit) {
  logger("The server is about to shutdown. There's no way back.")
  println("Code to shutdown the application here...")
}
```

它接受一个参数——logger，这个参数是一个函数，将一个字符串（m）作为输入值（注意，并没有使用这个字符串参数）且什么都不返回（返回类型为Unit）。在你关闭服务器之前，函数shutdown()调用传入的函数logger，但对函数logger的实现细节一无所知，而只知道它将一个字符串（其中包含要写入到日志中的消息）作为参数，且不返回任何值。

调用函数shutdown()时，调用者可传入一个lambda——直接传入函数logger的函数体：

```
shutdown({ msg: String -> println("Logged message: '$msg'") })
```

这里的lambda只是将msg打印到控制台。

将一个函数传递给另一个函数的开销非常高。函数在内部被定义为对象；函数shutdown()调用函数logger()时，将在幕后做大量的工作，这将消耗一定的处理时间。由于lambda函数能够访问其所属类的成员，因此将把成员变量的副本传递给lambda函数。就当今而言，这通常不是大问题，但在速度至关重要时，Kotlin提供了一种巧妙的解决方案：内联函数。对于将lambda作为输入的函数（如前述示例中的shutdown函数），通过在它前面加上inline，可让Kotlin编译器重写代码，将lambda的实现复制到函数中，从而避免函数再调用lambda。

考虑到这可能听起来令人迷惑，我们来看一个示例——在函数shutdown()前面加上inline：

```
inline fun shutdown(logger: (m: String) -> Unit) {
  logger("The server is about to shutdown. There's no way back.")
  println("Code to shutdown the application here...")
}
```

对于这个调用lambda函数的内联函数，编译器会重写调用它的代码。例如，请看下述调用函数shutdown的代码：

```
fun closeConnectionsAndShutdown() {
  println("Code that shutdowns active connections omitted...")
  shutdown({ msg: String -> println("Logged message: '$msg'") })
}
```

编译器将把它重写为类似于下面这样：

```
fun closeConnectionsAndShutdown() {
  println("Code that shutdowns active connections omitted...")
  val msg = "The server is about to shutdown. There's no way back."
  println("Logged message: '$msg'")
  println("Code to shutdown the application here...")
}
```

相比于原来的shutdown，这个版本的执行速度更快。

9.5　Kotlin 过程性编程

虽然Kotlin是一种纯粹的OOP语言，但它也支持过程性编程。这意味着不同于Java和Scala，可在类外定义函数和变量（正如你在本书前面看到的，使用Scala REPL时，可以不将函数和变量放在类中，但使用独立编译器scala时，必须将它们放在类中）。

编写程序时，如果不使用Kotlin的REPL交互式shell，将使用Kotlin编译器；而使用这个编译器来编译源代码时，可将函数和属性放在源代码文件的顶层。本章前面都是这样做的：

```
fun function1 {
  println("function1 is running...")
}
var property1: String = "default value of property1"
```

然而，不能将可执行的代码放在源代码的顶层；可执行的代码必须位于函数中。要创建可使用命令java或kotlin执行的JVM应用程序，必须定义一个main()函数。当函数具有如下签名时，Kotlin编译器将把它编译成静态方法，使其可用作JVM入口函数：

```
fun main(args : Array<String>) {
  // 可执行的代码……
  function1()
  println(property1)
}
```

鉴于JVM平台总是使用类，Kotlin编译器在幕后将源代码编译成类文件，以充当源文件中代码的包装器。对于编译得到的类，这样给它命名：在源代码文件名后面加上Kt；因此，如果源代码文件名为CoolProject.kt且首行为package com.example，则生成的类的全限定名为com.example.CoolProjectKt。

请注意，Kotlin REPL不会自动执行方法main()。要运行这个示例，请将这些源代码保存到名为procedural_programming.kt的文件中，再在命令提示符（Windows）或终端中执行如下命令：

```
kotlinc-jvm procedural_programming.kt
kotlin Procedural_programmingKt
```

这些命令的作用如下。

- 编译器kotlinc-jvm将procedural_programming.kt编译为与JVM兼容的文件Procedural_programmingKt.class，其中包含Java字节码。
- 前面说过，这个类文件中的类名为Procedural_programmingKt。
- 命令kotlin是JVM命令java的快捷方式，它确保将Kotlin运行时库添加到类路径中。由于这个类包含入口函数main，因此JVM能够运行这个应用程序。

9.6 风格指南

Kotlin文档包含主题“编码约定”（Coding Conventions），其中最重要的规则如下。

- 有疑问时使用Java转换。
- 使用4个空格缩进。
- 用冒号分隔子类和超类时，在冒号前后都加上一个空格，如class X : Y()。
- 在变量声明中，冒号前后都不添加空格，如val x:Int。
- 对于不返回任何值的方法，不指定可选的返回类型Unit。

9.7　小测验

(1) 下面哪项正确地描述了Kotlin？

a) 它是一种静态类型的函数式编程语言，包含一些OOP功能。
b) 它是一种静态类型的OOP语言，包含一些函数式编程功能。
c) 它是一种动态类型的函数式编程语言，包含一些OOP功能。
d) 它是一种动态类型的OOP语言，包含一些函数式编程功能。

(2) Kotlin允许继承多个父类吗？

a) 允许。在Kotlin中，一个类可扩展任意数量的类。
b) 不允许。在Kotlin中，一个类最多只能扩展一个类。

(3) 声明类时，如果没有显式地指定超类，其超类将是哪个类？

a) 它将没有超类；
b) 其超类将为`java.lang.Object`。
c) 其超类将为`kotlin.Object`。
d) 其超类将为`kotlin.Any`。

(4) 下面的代码能够在Kotlin REPL中运行吗？如果不能，请说明原因。

```
var k: Int = null
```

a) 能。运行这些代码时不会出现任何错误。
b) 不能，将引发错误，因为Kotlin的类型系统禁止将`null`赋给`Int`变量。
c) 不能，因为对于使用`var`定义的可变变量，不能将其初始化为`null`。
d) Kotlin使用关键字`nil`而不是`null`来表示空引用。

(5) 下面哪项不能在源代码文件的顶层声明？

a) 函数。
b) 属性。
c) 可执行的代码。
d) 以上答案都对。

9.8　小结

本章首先从Kotlin官网主页下载并安装了它。我们研究了REPL，并使用它来学习了Kotlin的一些基础知识，如定义函数和变量。我们很快就发现，Kotlin有很多与Java类似的功能，但使用

起来容易得多——通常需要编写的样板代码少得多。我们学习了Kotlin独特的类型系统，它在处理null引用方面尤其独特。我们学习了最重要的OOP主题，如定义类、给类添加函数和属性，以及给类添加JVM入口函数。我们还讨论了一些高级功能，如单例对象和伴生对象、数据类以及lambda函数。最后，你了解到Kotlin也可用于过程性编程，还了解了Kotlin的编码约定。

下一章将使用Oracle高级桌面GUI框架JavaFX创建一个Kotlin项目，并使用流行的构建系统Apache Maven JVM来构建这个项目。

第 10 章

Kotlin编程

10

在本章中，我们将使用工具包JavaFX编写一个小型的Kotlin桌面GUI应用程序。前一章使用的主要是Kotlin REPL，而本章将使用Eclipse IDE来编写代码。与Scala和Clojure编程一样，这里也需要安装一个插件；这个插件可在Eclipse Marketplace中找到，因此安装起来易如反掌。

至于构建工具，我们将使用Apache Maven。它最初是一个Java构建工具，但可使用插件对其进行扩展，以支持其他语言，如Kotlin。Apache Maven通过读取一个XML文件来构建项目，这个文件定义了各个阶段要使用的所有依赖和插件，还有构建过程的目标。我们将使用Kotlin小组提供的一个配置好的模板来创建这个项目。本章介绍如下主题：

- Eclipse IDE Kotlin插件；
- Apache Maven；
- 创建一个JavaFX桌面GUI应用程序。

10.1 Eclipse IDE Kotlin 插件

Kotlin是由JetBrains的一个开发小组开发的，这家公司开发了多款流行的商业IDE及其免费的社区版本，其中之一是JMV软件开发IDE IntelliJ IDEA，它提供了强大的Kotlin支持，这没什么可奇怪的。为了推广 Kotlin，JetBrains还开发了一个让Eclipse IDE能够支持Kotlin的插件，本章将使用这个插件。

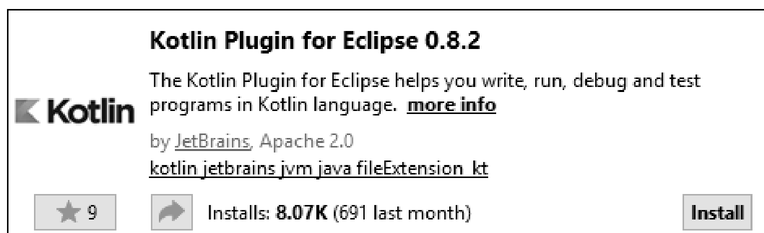
10.1.1 安装 Eclipse IDE Kotlin 插件

这个插件包含在Eclipse Marketplace中，因此安装起来简单得不能再简单了。编写本书期间，Eclipse Marketplace提供的该插件版本与官网的最新版本相同。现在，你应该对通过Eclipse Marketplace安装插件的过程很熟悉。

(1) 在Eclipse IDE中，选择菜单Help> Eclipse Marketplace...。

(2) 在文本框Find中输入`Kotlin`并按回车。

(3) 查找条目Kotlin Plugin for Eclipse（由JetBrains出品）并单击Install按钮：

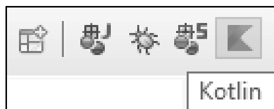


(4) 再按提示操作。Eclipse IDE将发出安全警告，并询问你是否要继续。如果你认为JetBrains没有给这个安装程序签名不是问题，请选择Yes确认要继续安装。另外，接受许可条款。最后，Eclipse IDE将询问你是否要立即重启Eclipse IDE，请单击Yes。

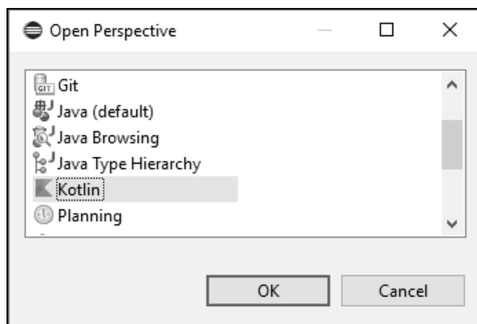
这就安装好了，现在Eclipse IDE能够支持Kotlin了。

10.1.2 切换到 Kotlin 透视图

Eclipse IDE Kotlin插件提供了独立的透视图，让你能够切换到针对Kotlin编程进行了优化的用户界面。为此，可在Eclipse IDE窗口右上角的工具栏中找到并单击工具提示为“Kotlin”的按钮：



如果找不到这样的按钮，请在这个工具栏中找到并单击工具提示为“Open Perspective”的按钮，这将打开一个对话框，其中列出了支持的所有透视图。请选择透视图Kotlin ，再单击OK按钮：



Eclipse IDE将显示针对Kotlin开发进行了全面优化的用户界面。

10

10.2　Apache Maven

JVM Kotlin开发人员通常使用JVM和Java领域常用的构建工具，其中的两个是Gradle 公司出品的Gradle和Apache Software Foundation出品的Maven，它们都能够管理依赖和构建项目。考虑到第4章开发小型Java应用程序时使用过Gradle，本章将使用Apache Maven。

Maven使用XML构建文件来构建项目，并严格地遵守"约定优先于配置"的范式。只要你遵守Maven的约定，就无需频繁地修改构建文件，但如果你想冲破藩篱，在构建项目中使用自定义动作，情况可能非常复杂甚至麻烦不断。大多数流行的插件都会在构建过程中添加新的功能或操作，但要找到满足需求的插件可能很难。就Kotlin开发而言，必须添加Kotlin Maven插件，让Maven知道如何编译Kotlin代码。

Eclipse IDE Kotlin插件不支持从GUI创建基于Maven的项目，但由于Eclipse IDE本身支持Maven，因此可手动创建一个项目，再将其导入到Eclipse IDE中。本节介绍如下主题：

- ❑ 安装Apache Maven；
- ❑ 下载预制的Kotlin基本套件（starter kit）；
- ❑ 在Eclipse IDE中导入项目。

10.2.1　安装 Apache Maven

鉴于基于JVM的既有项目大量地使用了Maven，因此志向远大的JVM开发人员最好安装这个工具，而不管以后是否在Kotlin项目中使用它。有关Maven的更详细信息，请参阅其主页http://maven.apache.org。

Maven的安装步骤与本书介绍的其他JVM工具很像。

(1) 访问Maven的项目主页以下载Maven。在这个主页的Download部分，从列表中找到附近的下载镜像，并下载最新的版本。编写本书期间，最新版本为3.5.0。Windows用户应下载ZIP文件（编写本书期间，这个文件名为apache-maven-3.5.0-bin.zip），而Linux和macOS用户应下载tar.gz归档文件（编写本书期间，这个文件名为apache-maven-3.5.0-bin.tar.gz）。
(2) 在你的系统中，将下载的归档文件解压缩到一个方便的目录中。
(3) 将其中的目录bin添加到环境变量`Path`中。

为了检查安装情况，在命令提示符（Windows）或终端窗口（macOS和Linux）中输入`mvn --help`并按回车，控制台窗口将出现一个长长的选项列表：

```
Command Prompt
C:\>mvn --help

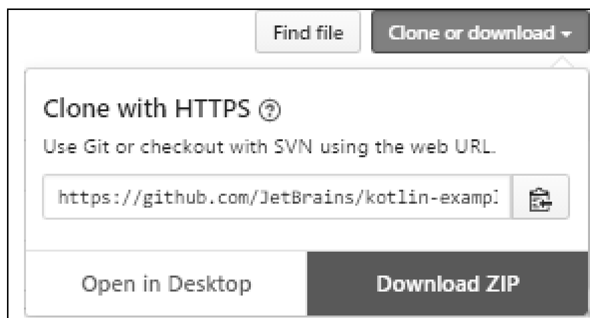
usage: mvn [options] [<goal(s)>] [<phase(s)>]

Options:
 -am,--also-make                        If project list is specified, also
                                        build projects required by the
                                        list
 -amd,--also-make-dependents            If project list is specified, also
                                        build projects that depend on
                                        projects on the list
 -B,--batch-mode                        Run in non-interactive (batch)
                                        mode
 -b,--builder <arg>                     The id of the build strategy to
                                        use.
 -C,--strict-checksums                  Fail the build if checksums don't
                                        match
 -c,--lax-checksums                     Warn if checksums don't match
```

10.2.2 下载预制的 Kotlin 基本套件

Kotlin开发小组提供了一个基本套件，其中包含一个可使用Gradle或Maven进行构建的项目。你可从Kotin小组的官方GitHub仓库下载这个基本套件：https://github.com/JetBrains/kotlin-examples。

通过使用你喜欢的浏览器访问这个GitHub页面，并单击Clone or download按钮，可下载一个ZIP文件，其中包含该仓库主分支的最新版本。请将这个文件解压缩到Eclipse workspace目录中。





如果你安装了Git，也可检出这个文件，方法是在命令提示符或终端窗口中切换到Eclipse的workspace目录，再执行命令`git clone https://github.com/JetBrains/kotlin-examples`。

下面通过编译并运行这个项目来检查Maven的安装情况。在命令提示符（Windows）或终端窗口（macOS和Linux）中，切换到目录kotlin-examples/maven/hello-world，并执行如下命令：

```
mvn compile
```

这将启动Maven并执行构建文件中的`compile`目标（goal）。Maven将下载编译这个项目所需的依赖项（其中包括Kotlin Maven插件），开始编译项目源代码文件，并指出成功地完成了这个任务。在这个过程中，Maven创建了一个target目录，该目录包含子目录classes，而这个子目录包含编译得到的类文件。

Maven基本套件中的构建文件经过了配置，使得使用Maven也可启动它。为了运行这个项目，请执行如下命令：

```
mvn exec:java
```

虽然这是一个Kotlin项目，但也可使用Maven的默认执行任务`java`，因为普通JVM命令`java`用于启动应用程序：

```
Command Prompt
[INFO] ------------------------------------------------------------------------
[INFO] Building hello-world 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) > validate @ hello-world >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) < validate @ hello-world <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ hello-world ---
Hello, world!
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.270 s
[INFO] Finished at: 2017-03-29T00:28:31+02:00
[INFO] Final Memory: 8M/184M
[INFO] ------------------------------------------------------------------------
C:\Kotlin\kotlin-examples\maven\hello-world>
```

除Maven的输出外，你还应看到问候语“Hello, World”。

10.2.3　在 Eclipse IDE 中导入项目

在JVM领域，Maven可谓家喻户晓，因此所有的Java版Eclipse IDE本身都支持Maven。因此，在Eclipse IDE中导入这个项目易如反掌。

(1) 在Eclipse IDE中，右击Package Explorer的空白区域并选择Import...。
(2) 在出现的Import对话框中，选择Projects from Folder or Archive并单击Next按钮。
(3) 单击文本框Import Source旁边的Directory按钮，切换到Eclipse的workspace目录（通常在你的用户主目录中），找到目录kotlin-examples-master并切换到其子目录maven，再选择目录hello-world并单击OK按钮。
(4) 最后，单击Finish按钮导入这个项目。

由于Eclipse IDE熟知Maven指定的约定，因此能够妥善地导入项目，并将最重要的Maven任务映射到正确的GUI元素。下面来核实它是否像期望的那样工作。在Package Explorer中，展开项目hello-world，并打开目录src/main/kotlin中的文件Hello.kt。将问候语“Hello, world!”修改为其

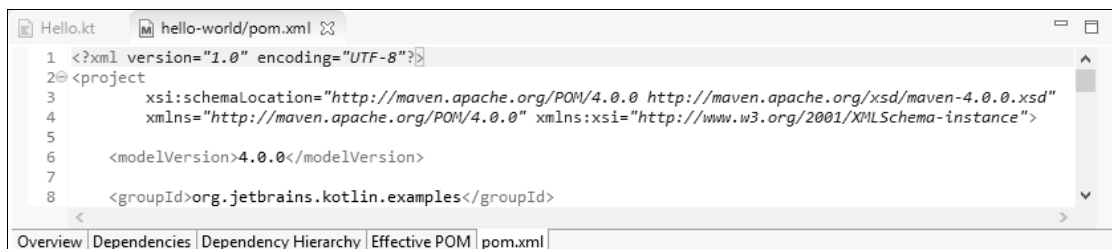
他内容，将文件存盘，再单击工具栏中工具提示为“Run Hello.kt”的按钮：



你将在Eclipse的Console选项卡中看到修改后的消息，这说明Eclipse IDE能够编译文件Hello.kt并运行其中的函数`main()`。

10.2.4 探索构建文件 pom.xml

接着往下介绍前，先来看看Maven构建文件——通常名为pom.xml。在Package Explorer中，打开项目hello-world的文件pom.xml。默认将显示一个概述（overview）页面，但我们要查看的是XML文件本身，因此请单击概述页面所在窗口底部的标签pom.xml，这将显示原始XML文件：





POM是Project Object Model（项目对象模型）的首字母缩写。从Maven 2开始，使用的就是POM文件格式的最新版4.0.0。

下面来讨论这个文件中的一些重要元素。

```
<project
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd"
 xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   ...
</project>
```

这是POM的根节点。前面说过，最新的POM版本为4.0.0。

```
<groupId>org.jetbrains.kotlin.examples</groupId>

<artifactId>hello-world</artifactId>

<version>1.0-SNAPSHOT</version>
```

groupId应为标识项目的字符串，它必须遵守JVM包名约定。artifactId指出了创建的JAR文件的文件名，但不包含版本号。版本是使用version元素定义的。

```
<properties>
  <kotlin.version>1.0.3</kotlin.version>
  <junit.version>4.12</junit.version>
  <main.class>hello.HelloKt</main.class>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

属性为键–值对，其中键是由元素名定义的，而值由内容定义。根据约定，以属性的方式定义依赖项的版本号，这样更新版本时只需修改一个属性，而无需查找并替换多个XML元素。

```
<dependencies>
  <dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-stdlib</artifactId>
    <version>${kotlin.version}</version>
  </dependency>
  ...
</dependencies>
```

元素<dependencies>内的<dependency>元素定义了项目的各个依赖项，其中一个依赖项是Kotlin标准库（kotlin-stdlib），它使用属性kotlin.version来指定版本。

```
    <build>
      <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>

<testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirector
y>
      <plugin>
        <artifactId>kotlin-maven-plugin</artifactId>
        <groupId>org.jetbrains.kotlin</groupId>
        <version>${kotlin.version}</version>
        ...
      </plugin>
      ...
</build>
```

这个构建文件的大部分内容都位于标签<build>和</build>之间，其中定义了多个插件，这里显示的是插件kotlin-maven-plugin，它让Maven能够支持Kotlin。Maven包含多个阶段（phase），而每个Maven插件都可修改阶段以及定义目标。

目标是在从命令行执行命令mvn时指定的；例如，当你从命令行运行命令mvn compile时，指定的目标为compile。

10.2.5　在 Eclipse 中更新构建文件

编写本书期间，Kotlin 1.1已经推出，但这个基本套件指定的Kotlin版本为1.0.3。为了修改Kotlin

版本，我将属性kotlin.version的值从1.0.3改为1.1.0：

```
<properties>
  <kotlin.version>1.1.0</kotlin.version>
  ...
</properties>
```

默认情况下，Kotlin编译器编译得到的是Java 1.6字节码，因此推荐添加下面的属性，让编译器生成更高效的Java字节码：

```
<properties>
  <kotlin.version>1.1.0</kotlin.version>
  <kotlin.compiler.jvmTarget>1.8</kotlin.compiler.jvmTarget>
  ...
</properties>
```

Eclipse IDE存在的一个问题是，如果在Maven构建文件中没有显式地指定Java编译器，它将默认使用Java 1.5编译器。由于我们的项目将使用这个Java编译器，因此这其实不是问题。



不幸的是，这意味着在开发这个项目的过程中，对于主项目和测试资源，Problem选项卡中将显示警告“Build path specifies execution environment J2SE-1.5...”。

编辑文件pom.xml后，必须让Eclipse刷新项目。为此，可右击项目名，并选择Maven>Update Project，再在出现的对话框中单击OK按钮。

10.3 创建 JavaFX 桌面 GUI 应用程序

我们将使用桌面工具包JavaFX GUI来创建一个简单的Kotlin桌面应用程序。JavaFX是最新的GUI工具包，大多数流行的**Java运行时环境**（Java Runtime Environment，JRE）版本都提供了它。用于Windows、macOS和基于桌面的Linux的Java版本都提供了JavaFX；最初，随Raspberry Pi的Raspbian操作系统预安装的Java SE Embedded 8也提供了JavaFX，但Oracle在较近的更新中将其删除了。另外，Solaris用户也无法使用JavaFX。



Oracle以开源许可的方式提供用于Java SE Embedded的JavaFX的源代码，因此高级用户依然能够在Raspberry Pi上编译并运行基于JavaFX的应用程序，但这种使用JavaFX的方式不在本章的探讨范围内。

强烈建议你在开发本章的项目时随时参阅JavaFX文档，这个文档可通过如下链接找到：http://docs.oracle.com/javase/8/javase-clienttechnologies.htm。

Kotlin非常适合用于JavaFX开发。由于Kotlin能够通过属性名访问类的属性，因此在Kotlin中，你无需调用获取/设置函数，而是可以像访问字段一样访问属性，这让代码整洁得多。但在幕后，Kotlin还是会调用设置/获取函数。下面是一个使用Java编写的简单示例：

```
// Java代码示例，请不要在Eclipse中输入它们

@Override
public void start(Stage stage) {
  stage.setTitle("Kotlin JavaFX Demo")
  stage.setScene(new Scene(new Pane(), 300.0, 300.0))
}
```

使用Kotlin编写时，代码如下：

```
// Kotlin代码示例，请不要在Eclipse中输入它们

override fun start(stage: Stage) {
  stage.title = "Kotlin JavaFX Demo"
  stage.scene = Scene(Pane(), 300.0, 300.0)
}
```

虽然没什么大不了，但你可能也会认为Kotlin代码要整洁、易读些。所幸Eclipse IDE Kotlin插件会在自动补全建议中显示属性，这一点你稍后就会看到。

10.3.1　定制项目

我们将对前一节在Eclipse IDE中导入的项目进行定制。为此，请执行如下操作。

(1) 将目录src/test/kotlin中的文件HelloTest.kt删除：右击这个文件并选择Delete...。
(2) 将目录src/main/kotlin中的文件Hello.kt重命名为App.kt：在Package Explorer单击文件Hello.kt，并按F2。
(3) 打开文件pom.xml，找到<properties>部分，将其中的属性`<main.class>hello.HelloKt</main.class>`改为`<main.class>javafxdemo.AppKt</main.class>`。这个类将包含程序的JVM入口方法`main()`。
(4) 刷新这个Maven项目：右击项目名并选择Maven>Update Project。

10.3.2　创建可运行的应用程序

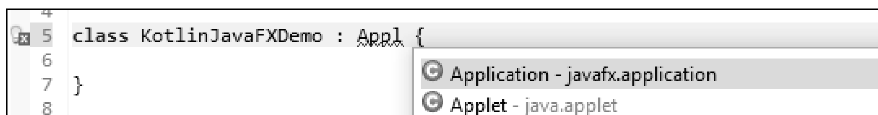
打开目录src/main/kotlin中的文件App.kt，将其内容替换为如下代码：

```
package javafxdemo
fun main(args: Array<String>) {
}
```

将光标放在package语句后面并添加一个空行，再输入如下代码：

```
class KotlinJavaFXDemo :
```

再输入`Appl`并按Ctrl + 空格（Apple机上为cmd + 空格），Eclipse IDE将打开一个包含建议的弹出窗口：



在列表中找到并双击javafx.application包中的Application类，也可单击并按回车键。这将自动输入类名Application，并添加相应的import语句。接下来添加()，让这个子类自动调用父类javafx.application.Application的不接受任何参数的构造函数。现在的代码应类似于下面这样：

```
package javafxdemo

import javafx.application.Application

class KotlinJavaFXDemo : Application() {
}

fun main(args: Array<String>) {
}
```

Application是JavaFX工具包中的一个类，这是一个抽象类，包含多个具体方法和一个抽象方法，其中的抽象方法的Java定义如下：

```
// Java代码
public abstract void start(Stage primaryStage)
```

基于JavaFX的应用程序必须扩展Application类，并实现其方法start()，因此我们必须在KotlinJavaFXDemo类中提供这个方法的实现。将光标放在KotlinJavaFXDemo类的定义中，按Ctrl + 1（在macOS中为cmd + 1）并双击Implement Members：





Ctrl + 1（在macOS中为cmd + 1）是在Eclipse IDE命令Quick Fix（快速修复）的快捷键，可快速修复常见问题。

现在这个类应类似于下面这样：

```
class KotlinJavaFXDemo : Application() {
  override fun start(primaryStage: Stage?) {
    TODO()
  }
}
```

遗憾的是，这并不会自动添加导入Stage的语句；有鉴于此，将光标放在类名Stage?中的

问号前面，并按Ctrl +空格（在macOS中为cmd +空格），再选择`javafx.stage`包中的`Stage`类。这将添加所需的`import`语句。

将类名`Stage`后面的问号删除（因为这个方法是JavaFX提供的，不可能为`null`），再将函数`start()`的实现替换为如下代码。输入每个类名时，都请使用快捷键Ctrl＋空格（macOS中为cmd＋空格）启动自动补全功能，并选择`javafx`包中的类：

```
override fun start(primaryStage: Stage) {
    primaryStage.title = "Kotlin JavaFX Demo"
    val pane = Pane()
    val scene = Scene(pane, 500.0, 500.0)
    primaryStage.scene = scene
    primaryStage.show()
}
```

这些代码实现的功能很多，下面来详细说说。

- `Stage`对象是JavaFX应用程序的顶级容器。JavaFX默认自动创建一个主窗口，并将指向这个窗口的引用（一个`Stage`对象）传递给方法`start`。
- 我们将这个主窗口的title属性（标题）改为Kotlin JavaFX Demo。默认情况下，窗口的标题为空。
- 创建了一个`Pane()`对象。`Pane`对象可以有子对象，其中每个子对象都可以是GUI元素，在`Pane`对象渲染自己时会被自动渲染。
- 我们创建了一个`Scene`对象。`Scene`对象包含根对象，而根对象包含所有要绘制的子对象。就这里而言，这好像毫无用处，因为这里只有一个`Pane()`对象，但稍后你就会看到，窗口通常包含多个对象。我们告诉JavaFX，我们希望这个场景（scene）的高度和宽度都为500像素。
- 我们将这个`Scene`对象赋给窗口`primaryStage`。知道所需的尺寸后，JavaFX创建一个500像素 × 500像素的对话框。
- 最后，让窗口`primaryStage`可见。



JavaFX通常使用基本类型`double`变量（而不是整数）来指定位置和尺寸。

我们还需要实现JVM入口函数`main()`。要启动JavaFX应用程序，必须调用`Application`类的公有静态方法`launch`。请将函数`main()`的实现替换为如下代码：

```
fun main(args: Array<String>) {
    Application.launch(KotlinJavaFXDemo::class.java)
}
```

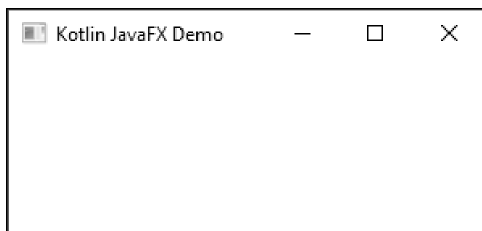


请确保这个函数没有放在类中，否则它将成为普通的实例方法。仅当这个方法不在任何类中时，Kotlin才会将它编译为可用作JVM入口方法的静态方法。

方法launch是Application类的一个静态方法，在Kotlin中，这意味着只能通过类名Application来访问它。调用这个方法时，必须指定一个这样的参数，即指向扩展Application类的Java类的引用；在Kotlin中，这是通过将它转换为class.java对象实现的。

前一章说过，当一个方法放在顶层（不在任何类中）时，Kotlin将创建一个类，这个类的名称由源代码文件名和后缀Kt组成。因此，就这里而言，方法main将放到javafxdemo包的AppKt类中，这个类也是我们前面修改构建文件pom.xml时指定的主类。

现在应该能够运行这个应用程序了，为此可按Ctrl + F11（在macOS中为cmd + F11），也可单击工具栏中的启动按钮。程序启动后，将出现一个空窗口：



这不太令人激动，下面着手在窗口中打印一些内容。

10.3.3 编写扩展函数

Kotlin提供了一种有趣的功能——扩展函数。扩展函数是与特定类型相关联的函数。与扩展函数相关联的类型称为接受类型（receiver type），是扩展函数将添加到其中的类。接受类型的实例可像调用其其他方法一样调用扩展函数，唯一的差别是必须先导入扩展函数才能调用它。无需通过继承就可添加扩展函数，因此可给任何类添加扩展函数，包括closed（Java中为final）类。下面给JavaFX类Pane添加一个扩展函数，以令人眼花缭乱的方式打印一条消息。

在Package Explorer中，右击目录src/main/kotlin并选择New>Kotlin File...。在文本框Package中输入javafxdemo.extensions，并在文本框Name中输入PaneExtensions，再单击Finish按钮。这将创建新文件，其中包含必要的package语句。在这个文件中添加如下函数：

```
import javafx.scene.layout.Pane
fun Pane.prettyPrint(y: Double, text: String) {
}
```

通过在函数名前面加上类名Pane和句点，让Kotlin知道这是一个扩展函数，Pane类及其子类的所有实例都可调用它。导入这个扩展函数（我们稍后将这样做），就可像下面这样调用它：

```
// 示例代码（请不要在Eclipse中输入它们）
val pane = Pane()
pane.prettyPrint(50.0, primaryStage.title)
```

在调用prettyPrint的代码看来，prettyPrint就像是Pane类的一个普通方法一样。在函数Pane.prettyPrint()中，可访问用来调用它的Pane实例，为此只需使用this引用即可。明白这些后，我们来实现扩展函数prettyPrint。



别忘了使用Ctrl +空格（在macOS中为cmd + 空格）来输入所有的类名，以自动添加必要的import语句；另外，务必选择javafx包中的相应类。

扩展函数prettyPrint的代码如下：

```
fun Pane.prettyPrint(y: Double, text: String) {
    val t = Text()
    t.text = text
    t.font = Font.font("Verdana", FontWeight.BOLD, 30.0)
    t.fill = Color.DARKBLUE

    t.x = 0.0
    t.y = y
    this.children.add(t)
}
```

为让你能够从正确的包中选择类（或手动编写正确的import语句），下面列出了上述代码中用到的所有类的全限定类名：

- javafx.scene.layout.Pane;
- javafx.scene.text.Text;
- javafx.scene.text.Font;
- javafx.scene.text.FontWeight;
- javafx.scene.paint.Color。

打开目录src/main/kotlin中的文件App.kt，并添加如下import语句：

```
import javafxdemo.extensions.prettyPrint
```



注意，在扩展函数的全限定名中，并不包含源代码文件名PaneExtensions，这是因为在Kotlin中，包名可以与源代码文件的目录结构完全不同。

如果我们要给这个文件中的其他类添加扩展函数prettyPrint，这完全可行；在这种情况下，只需前述import语句，就可调用所有名为prettyPrint的扩展函数。

打开文件App.kt，在JavaFXDemo类的方法start()中，将如下代码行添加到代码行val scene = Scene(pane, 500.0, 500.0)的后面：

```
pane.prettyPrint(100.0, primaryStage.title)
```

现在方法start()应类似于下面这样：

```
override fun start(primaryStage: Stage) {
    primaryStage.title = "Kotlin JavaFX Demo"
    val pane = Pane()
    pane.prettyPrint(50.0, primaryStage.title)

    val scene = Scene(pane, 500.0, 500.0)
    primaryStage.scene = scene
    primaryStage.show()
}
```

再次运行这个程序，它应使用很大的字体打印文本`Kotlin JavaFX Demo`：



作为最后的修饰，我们来对打印的消息应用一种效果。为此，打开文件PaneExtensions.kt，并在代码行`this.children.add(t)`前面添加如下代码：

```
val shadow = InnerShadow()
shadow.offsetX = 2.0
shadow.offsetY = 2.0
t.effect = shadow
```

为了让这种效果更突出，将代码行`t.fill = Color.DARKBLUE`改为`t.fill = Color.YELLOW`。

再次运行这个程序，现在它看起来要漂亮些：



10.3.4 布局窗格

JavaFX有多个内置的布局窗格（layout pane）类。可像前一个示例那样使用`Pane`类来手动放置每个子控件——通过计算它们的X和Y坐标，但这样做很繁琐，在窗口包含多个控件且可调整大小时尤其如此。

在大多数情况下，更佳的解决方案是使用布局窗格类，这些类负责自动放置子控件以及调整它们的大小。下表列出了最重要的内置布局。

布　局　类	描　　述
BorderPane	提供5个子窗格：上、左、中、右和下
HBox	在一个水平框中将每个子控件都放在前一个右边
VBox	在一个垂直框中将每个子控件都放在前一个下方
StackPane	将每个控件都堆叠在前一个上面，从而将控件组合起来
GridPane	以行列方式创建一个类似于网格的结构
FlowPane	将每个控件都放在前一个右边，到达最大宽度后重起一行，并将下一个控件放在这行的第一列。也可按先列后行的方式依次放置控件
TilePane	类似于FlowPane，但每个子控件的尺寸都相同
AncorPane	支持将子控件锚定在固定的位置（上、下、左、右或中）；调整大小时，确保子控件相对于锚点的位置不变

由于所有的布局窗格都是Pane的子类，因此可混合使用不同的布局窗格。本章后面将演示一些较为常用的布局窗格。



有关布局的完整信息，请参阅Oracle JavaFX官方手册的Work with Layouts部分，其网址为http://docs.oracle.com/javase/8/javafx/layout-tutorial/。

10.3.5　实现基于 BorderPane 的布局

接下来在窗口中添加非常简单的动画。为此，首先来创建一个BorderPane，它将作为场景的根窗格。我们将把当前的Pane（它包含函数prettyPrint的输出）赋给BorderPane的top子窗格，让其充当应用程序的彩色标题。

打开文件App.kt，并重写函数start：使用下面的代码替换原来的代码。与往常一样，对于还未导入的类，别忘了使用Ctrl + 空格（或cmd + 空格）来输入它们的名称：

```
override fun start(primaryStage: Stage) {
    primaryStage.title = "Kotlin JavaFX Demo"
    val textField = TextField()
    val mainPane = BorderPane()
    mainPane.top = createHeaderPane(primaryStage.title)

    val scene = Scene(mainPane, 500.0, 500.0)
    primaryStage.scene = scene
    primaryStage.show()
}
```

我们对代码进行了重构：对于表示标题的Pane实例，我们将创建它的代码放在一个独立的函数中，这个函数将在稍后编写。我们创建了一个TextField实例，用于放置用户输入的文本。还创建了一个名为mainPane的实例，并将其top子窗格设置为headerPane。现在，场景的根节点为mainPane。

下面来编写函数createHeaderPane()。将其放在函数start()的后面，并包含如下代码：

```
fun createHeaderPane(title: String): Pane {
    val pane = Pane()
    pane.prettyPrint(30.0, title)
    return pane
}
```

运行这个项目，会发现没有什么变化。然而，拜BorderPane实例所赐，我们现在能够轻松地在窗口左边、中间、右边和底部添加新窗格了。

下面在窗口底部添加一个文本输入框，让用户能够输入将在屏幕上移动的文本。首先来编写一个函数，用于创建包含一个标签和文本框的HBox（水平框）。请将这个函数放在函数createHeaderPane()的后面。



对于还未导入的类，别忘了使用Ctrl＋空格（或cmd＋空格）来输入它们的名称，并选择javafx包中的相应类。为了节省篇幅，本章后面不再提醒你这样做。

同样，我们将在一个独立的函数中创建并设置HBox窗格。为此，在前一个函数后面添加如下代码：

```
fun createInputPane(textField: TextField): Pane {
    val label = Label("Input text:")
    label.minWidth = 65.0

    val inputPane = HBox()
    inputPane.children.add(label)
    inputPane.children.add(textField)

    HBox.setHgrow(textField, Priority.ALWAYS)
    return inputPane
}
```

下面来详细说说这些代码。

- 将一个文本输入框（在JavaFX中这种控件名为TextField）作为参数传递给了这个函数。之所以这样做，是因为调用各种函数时，需要提供指向这个对象的引用；在这里，我们需要使用这个引用将文本框添加到HBox中。
- 我们创建了一个尺寸固定（宽65像素）的标签。
- 接下来，我们创建了一个HBox布局窗格。前面说过，HBox布局窗格将其子节点（控件）依次放置在一行中。
- 将标签和textField控件都添加到了HBox类实例inputPane的子控件列表中。
- 通过调用HBox类的静态方法setHGrow，我们让HBox知道应让textField控件占据其余下的所有空间。

- 请注意，HBox是Pane的子类，因此我们将函数createInputPane()返回类型指定为Pane而不是HBox。以后如果需要修改这个函数的实现，可保持其返回类型不变。

下面来将这个HBox添加到BorderPane中，否则将无法看到函数createInputPane()的效果。为此，向上滚动到函数start()，并在其中的代码行mainPane.top = createHeaderPane(primaryStage.title)后面添加如下代码：

```
mainPane.bottom = createInputPane(textField)
```

现在再次运行这个应用程序，你将在窗口底部看到一个标签和一个文本框：



如果你尝试调整窗口的大小，将发现标签和文本框依然位于窗口底部（这是因为当你调整窗口的高度时，LayoutPane会自动移动其底部部分），同时文本框的宽度总是窗口宽度减去标签宽度（这是因为我们调用了HBox类的静态方法setHGrow）。

10.3.6　实现动画

我们要让用户输入的文本在屏幕上移动。为了实现这种动画，需要跟踪一些值，因此我们将创建一个处理动画的类。不同于Java，Kotlin支持在同一个源代码文件中定义多个类，而不管这些类使用的访问限定符是什么。请在KotlinJavaFXDemo类的后面添加如下代码：

```
class AnimatedText {
  val animatedText = Text()
  val animationPane = Pane()
  var directionX = 3.0
  var directionY = 3.0

  fun getPane(textField: TextField): Pane {
    animatedText.x = 0.0
    animatedText.y = 0.0
    animatedText.font = Font.font("Verdana", FontWeight.BOLD, 15.0)
    animationPane.children.add(animatedText)
    return animationPane
  }
}
```

通过将javafx.scene.text.Text对象的x和y属性都初始化为0.0，让文本一开始位于变

量animationPane表示的窗口的左上角（0,0）。变量directionX和directionY用于实现动画：每一步都使用它们将文本沿水平和垂直方向都移动3像素。请注意，我们将包含用户输入的TextField控件传递给了函数getPane()。

现在，我们需要创建一个每帧都调用的函数，它计算文本的新位置并在必要时调整移动方向。为了让你能够完成这种任务，JavaFX提供了AnimationTimer类，这是一个抽象类，包含一个在每帧中都将调用的抽象方法——handle()。这个方法将帧的时间（一个表示上次调用后过去了多长时间的long值）作为参数，但为简单起见，我们没有在这个示例中使用它。如果你要在实际应用程序中实现逼真的动画，必须使用这个值来计算上次调用后过去了多长时间，并相应地移动物体。

我们在AnimatedText类中实现AnimationTimer类，为此将光标放在代码行var directionY = 3.0的末尾，添加一个空行，再添加如下代码：

```
val timer = object : AnimationTimer() {
  override fun handle(now: Long) {
    if (animatedText.x < 0.0 || animatedText.x > animationPane
        .width - animatedText.layoutBounds.width)
      directionX = -directionX

    if (animatedText.y < 0.0 || animatedText.y > animationPane
        .height)
      directionY = -directionY
      animatedText.x += directionX
      animatedText.y += directionY
  }
}
```

语法object : AnimationTimer()可能看起来有点怪异。前面说过，AnimationTimer是一个抽象类，这意味着不能直接实例化它，因为扩展它的类必须提供它的抽象方法的实现。在刚才所说的代码行中，我们创建了一个扩展抽象类AnimationTimer的匿名对象。注意，我们没有给这个类指定名称，而是只指定了要将指向这个对象的引用存储到引用变量timer中。由于AnimationTimer类唯一的抽象方法是handle，因此匿名类只需重写这个方法。在这个方法中，我们检查文本的当前位置，并决定是否要在X和Y轴上沿相反的方向移动，再相应地更新对象animatedText的X和Y位置。

必须启动这个定时器（timer），我们在AnimatedText类的函数getPane()中这样做。为此，在代码行animationPane.children.add(animatedText)的后面，添加如下代码行：

```
timer.start()
```

我们需要将这个窗格加载到主场景的BorderPane布局的一个子窗格中，为此向上滚动到函数start()，并在其中的代码行mainPane.bottom = createInputPane(textField)后面添加如下代码行：

```
mainPane.center = AnimatedText().getPane(textField)
```

如果你现在运行这个应用程序，将发现没有任何变化。另外，当你在文本框中输入文本后，也不会发生任何事情。为什么会这样呢？因为我们还没有处理`TextField`中的输入，也没有在对象`animatedText`中以硬编码的方式指定文本。现在该来改变这种状况了。如果你使用的是当前市面上流行的其他的GUI工具包，这意味着需要给输入控件`TextField`添加一个侦听函数，再将其输入传递给对象`animatedText`，但JavaFX提供了一种强大得多的功能。

在JavaFX中，可将一个控件的属性绑定到另一个控件的属性。这种绑定可以是单向的（这意味着如果属性A被绑定到属性B，属性A将在它发生变化时自动更新属性B，但反过来不会），也可以是双向的（如果绑定了属性A和B，其中任何一个属性发生变化时，都会更新另一个属性）。显然，双向绑定的开销很高，且在这个示例中不需要双向绑定，因为在这个应用程序中，只有TextField控件能接受变化。为了将`textField`控件的属性`text`绑定到控件`animatedText`，在`AnimatedText`类的`getPane()`方法中，在代码行`return animationPane`前面添加如下代码行：

```
animatedText.textProperty().bind(textField.textProperty())
```

这样，当`textField`的`text`属性发生变化时，`animatedText`的`text`属性将自动更新，但反过来不会。为防范这一点，JavaFX将在你通过代码修改`animatedText`的属性`text`时引发异常。

请运行这个应用程序。当你在文本框中输入文本后，这些文本将在窗口内来回移动：



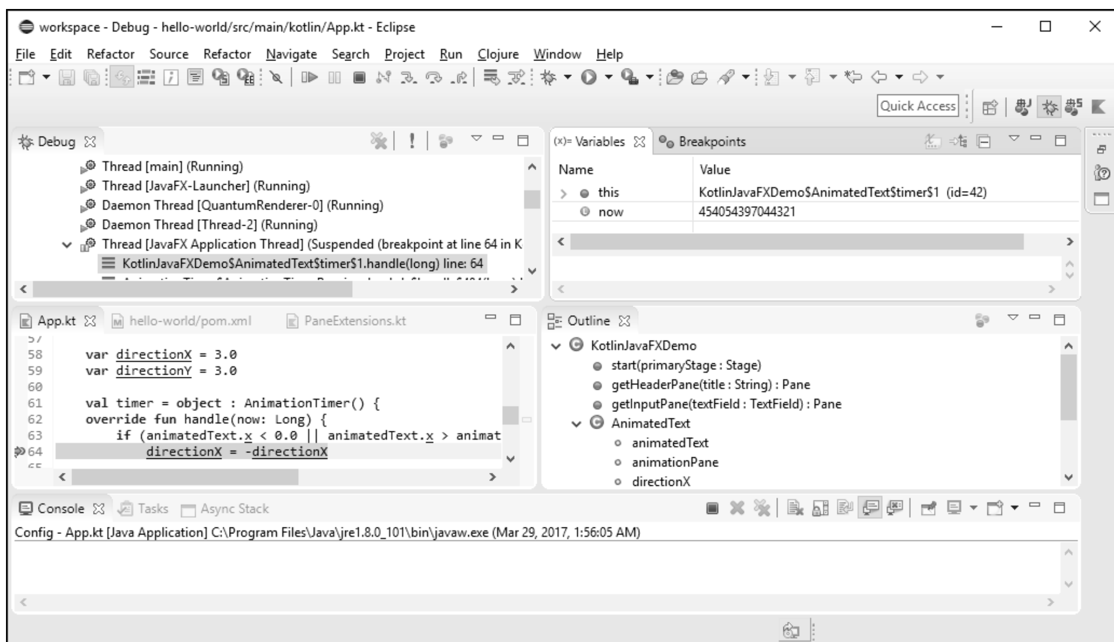
如果你输入很长的文本，或者在文本移到窗口右边时缩小窗口，将发现文本可能不再沿水平方向移动，甚至不见了（当你缩小窗口后，它依然停留在原来的区域附近）。

下面来尝试找出导致这种问题的原因。请停止运行这个程序，我们将使用调试器来找出导致这种问题的原因。

10.3.7　调试程序

选择Eclipse菜单Run>Debug或按F11，程序将在调试模式下运行。输入一些文本，将窗口增大，等文本移到窗口右边后迅速缩小窗口，尝试让文本消失或停止不动。

出现这种情况时，使用调试器来尝试找出其中的原因：



按下面的说明打开并使用调试器。

(1) 让程序继续运行并返回到Eclipse IDE。

(2) 打开文件App.kt并找到代码行`directionX = -directionX`。在这行代码的左边有行号，请右击这个行号并在出现的菜单中选择Toggle Breakpoint。

(3) 执行到包含断点的代码行后，程序将停止执行，并询问你是否要切换到Debug透视图（Eclipse提供的专门用于调试的透视图）。请选择Yes，因为这样Eclipse将显示针对调试进行了优化的用户界面。

(4) 在Debug透视图中，左上角的窗口显示了当前应用程序中所有正在运行的线程，而右上角显示了当前函数的变量及其值。由于函数`handle`没有使用局部变量，因此你只能看到变量`this`（`AnimatedText`的实例变量）和`now`（函数`handle`的参数）。

(5) 在包含代码的窗口中，包含断点的代码行将呈高亮显示。

(6) 在工具栏中，找到并单击工具提示为“Resume”的按钮。

(7) 你将发现程序将再次在运行到断点时立即停止。单击Resume按钮多次，你将发现每次调用函数`handle`时，好像都修改了directionX。

(8) 为找出原因，在Variables选项卡中展开`this`变量，你将看到匿名的`AnimationTimer`实例的变量。

(9) 注意，其中包含变量`this$0`。展开它，它包含`AnimationTimer`实例的父对象（一个`AnimatedText`对象）的变量。展开条目`animatedText`，这将显示对象`animatedText`的所有属性。找到属性`x`并查看其值，你将发现它大于0。

由于`animatedText.x`大于当前窗口余下的空间，因此每次调用函数`handle`时都将改变方向，导致文本停止移动。在工具栏中，找到并单击工具提示为“Terminate”的按钮，再切换到Kotlin透视图：找到并单击Eclipse IDE窗口右上角的Kotlin按钮。

为了修复问题，可在修改移动方向时重置文本的位置。这样做，可确保文本在窗口内（除非窗口太小）。下面首先处理X方向。找到如下两行代码：

```
if (animatedText.x < 0.0 || animatedText.x > animationPane.width -
    animatedText.layoutBounds.width)
  directionX = -directionX
```

将上述代码替换为如下代码：

```
if (animatedText.x < 0.0) {
  animatedText.x = 0.0
  directionX = -directionX
} else if (animatedText.x > animationPane.width - animatedText
          .layoutBounds.width) {
  animatedText.x = animationPane.width - animatedText
                  .layoutBounds.width
  directionX = -directionX
}
```

对Y位置做同样的处理。找到如下两行代码：

```
if (animatedText.y < 0.0 || animatedText.y > animationPane.height)
  directionY = -directionY
```

将这些代码替换为如下代码：

```
if (animatedText.y < 0.0) {
  animatedText.y = 0.0
  directionY = -directionY
} else if (animatedText.y > animationPane.height) {
  animatedText.y = animationPane.height
  directionY = -directionY
}
```

再次运行这个应用程序。现在它稳定得多了，不会在你调整窗口大小或输入的文本太长时停滞不前。当文本不在窗口内时，它将向后移动，以便在窗口内（条件是窗口足够大）。

10.4　小结

本章创建了一个以简单动画方式移动文本的小型GUI桌面应用程序。我们首先安装了Eclipse IDE Kotlin插件，还安装了流行的JVM构建工具Apache Maven，它使用XML格式的构建文件。我们从Kotlin开发小组的GitHub页面下载了一个包含Maven构建文件的基本套件，并将其用作项目的模板。我们在Eclipse IDE中导入这个项目；由于Eclipse本身就支持Maven，因此我们无需做任何配置，Eclipse会自动将其GUI操作映射到正确的Maven目标。

就这样，我们终于为编写前述桌面GUI应用程序做好了准备。在开发这个应用程序的过程中，我们研究了各种JavaFX概念，还学习了一项新的Kotlin功能——扩展函数。我们在代码中遇到了bug，并使用调试器找到原因并修复了问题。

下一章将详细介绍Apache Groovy。不同于Kotlin，Groovy是一种动态类型语言，并提供了包含额外类的大型库。

第11章 Groovy

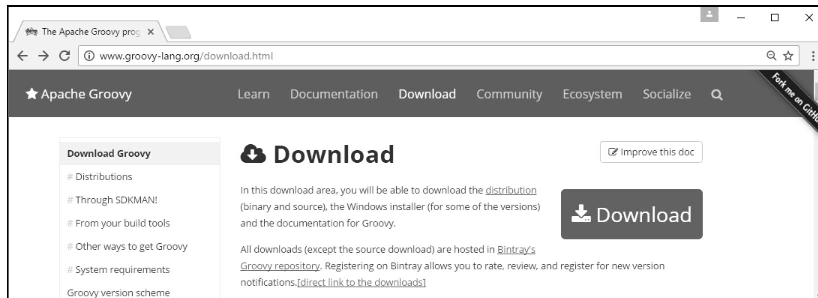
Groovy是较早的JVM语言之一，最初旨在在JVM中提供类似于Python的体验，这在当时是一种闻所未闻而神奇的想法。从本质上说，Groovy是一种动态类型语言，这意味着声明变量时无需指定类型，而方法调用是在运行阶段而不是编译阶段解析的，这提供了使用Java和Kotlin等静态语言难以实现的可能性。Groovy不同寻常，允许程序员在编译特定的类时将编译器切换到静态类型模式。在这种模式下，编译器将在编译阶段检查类型和方法调用，就像静态语言的编译器那样。

本章介绍如下主题：

- 安装Groovy；
- REPL shell GroovyConsole和GroovyShell；
- Groovy基础知识；
- 面向对象编程；
- Groovy开发包（GDK）；
- 动态和静态编程；
- 小测验。

11.1 安装 Groovy

相比于本书前面介绍的其他语言，Groovy的安装方式没有什么不同。使用你喜欢的浏览器，访问Groovy主页（http://groovy-lang.org）：



安装步骤如下。

- 在主页中找到Download部分。
- 单击显眼的Download按钮，这将下载一个ZIP文件。编写本书期间，这个文件名为apache-groovy-sdk-2.4.10.zip。
- 将这个文件解压缩到方便的地方，再将其中的bin目录添加到环境变量`Path`中。

Groovy自带了两种REPL环境：基于GUI的GroovyConsole和基于文本的GroovyShell。下面来检查安装情况：启动GUI应用程序GroovyConsole。按前面的指示将目录bin添加到环境变量`Path`中后，启动命令提示符（Windows）或终端（macOS和Linux），再执行如下命令：

```
GroovyConsole
```

这将启动GroovyConsole。这个程序适合运行小型Groovy脚本，本章都将使用它。

GroovyConsole 和 GroovyShell

前面说过，Groovy自带了两种REPL环境：

- GroovyConsole（桌面GUI应用程序）；
- GroovyShell（基于文本的shell）。

它们都可用来尝试运行本章的代码片段。

1. GroovyConsole

GroovyConsole是一个易于使用的桌面GUI应用程序，可用来交互式地编写和执行Groovy代码。这里只介绍其最常用的功能，但需要指出的是，它也提供了一些供高级用户使用的复杂功能。为启动GroovyConsole，请在命令提示符或终端窗口（macOS和Linux）中运行前述启动脚本：

```
GroovyConsole
File Edit View History Script Help
1 def book = "Introduction to JVM Languages"
2 def publisher = "Packt Publishing"
3 println "$book"
4 publisher

groovy> def book = "Introduction to JVM Languages"
groovy> def publisher = "Packt Publishing"
groovy> println "$book"
groovy> publisher

Introduction to JVM Languages
Result: Packt Publishing

Execution complete.    4:10
```

在窗口的上半部分可输入代码，而下半部分显示输出和其他相关的信息。



在很大程度上说，Groovy与Java语法兼容，因此在GroovyConsole和GroovyShell中，可输入大多数Java语句。但也存在一些不兼容的情况，这将在下一章介绍。

下面来尝试输入一段代码。为此，在窗口的上半部分输入如下代码：

```
def random = new Random()
random.nextInt(10)
```

按Ctrl + R（在macOS中为cmd + R），或在工具栏中找到并单击工具提示为“Execute Groovy Script”的按钮。



在窗口的下半部分，将打印被执行的代码，并最终打印一个位于0~9之间的数字。GroovyConsole总是打印最后评估的值。运行这个脚本多次，你将发现每次都在之前的输出后面显示新输出。

GroovyConsole有很多定制选项，下表介绍了菜单View中一些较常用的选项。

菜单View中的选项	描　　述
Clear output	清空输出窗口，在Windows和Linux中的快捷键为Ctrl + W，在macOS中的快捷键为cmd + W
Auto Clear Output On Run	被选中时，将在运行脚本时自动清空输出窗口；默认未选中
Show Script in Output	被选中时（默认被选中），将把脚本的代码打印到输出窗口
Show Full Stack Traces	被选中时（默认被选中），如果引发了异常，将在输出窗口中打印完整的栈跟踪。如果未选中，将只显示栈跟踪的最后一个条目
Detach output	被选中时（默认为未选中），将在不同的窗口中显示输入和输出

其他一些值得注意的功能如下。

- 通过按Ctrl + /（在macOS中为cmd + /），可取消对当前行或选定行的注释。
- 可将目录和/或JAR文件添加到类路径中，为此可选择菜单Script>Add Jar (s) to the ClassPath或Add Directory to the ClassPath。在你需要动态地探索外部库或工具包的API时，这项功能提供了极大的便利。
- 可保存和加载包含Groovy代码的脚本，为此可选择菜单File>Open或File>Save。

2. GroovyShell

GroovyShell是更传统的基于文本的REPL shell，类似于本书前面介绍的Scala、Clojure和Kotlin

的REPL。要启动它，可运行子目录bin中的启动脚本groovysh：

```
Command Prompt - groovysh.bat
Groovy Shell (2.4.10, JVM: 1.8.0_112)
Type ':help' or ':h' for help.
-------------------------------------------------------------------------------
groovy:000> i = 40
===> 40
groovy:000> j = i + 2
===> 42
groovy:000> :show variables
Variables:
  i = 40
  _ = 42
  j = 42
===> [i:40, _:42, j:42]
groovy:000>
```

在这个shell中，命令大都以冒号（：）打头。要获取可用的命令列表，可执行命令`:help`，也可使用快捷方式`?`。还可使用`:help`或`?`来获取特定命令的更详细信息。例如，要获取命令`:show`的更多信息，可执行命令`? :show`。要测试命令`:show`，可依次执行如下命令：

```
i = 40
j = i + 2
:show variables
```

这将显示所有的变量及其值。其中的_表示最后计算得到的值，这里为`42`。

要退出GroovyShell，可执行命令`:exit`或`:quit`（这两个命令都行，这是典型的Groovy风格）。

11.2　Groovy 语言

在很大程度上说，Groovy语言与Java语言兼容，因此对Java开发人员来说，Groovy学起来很容易。在Java中很多必不可少的元素在Groovy中都是可选的。Groovy使用的语义与Java相同，因此本章重点介绍Java和Groovy的不同之处。

Groovy的根本宗旨是紧凑、舒心和灵活。下面先来看一个简单的Java类：

```
class Person {
  private String name;
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public static void main(String[] args) {
    Person p = new Person();
```

```
    p.setName("fooBar");
    System.out.println(p.getName());
  }
}
```

这个类在Groovy中能够编译并运行，你可以在GroovyConsole中输入并执行这些代码。然而，通过利用Groovy特有的结构，可使用少得多的代码来编写这个程序：

```
class Person {
  String name
  static void main(String[] args) {
    def p = new Person()
    p.name = "fooBar"
    println p.name
  }
}
```

下面来详细说说这些代码。

- Groovy不要求代码行以分号结尾。
- 只需通过指定属性的类型和名称就可创建它（稍后你将看到，属性的类型也是可选的）。Groovy将自动创建一个私有变量以及公有的获取函数和设置函数，在这里它们分别是`name`、`getName()`和`setName()`。
- 在Groovy中，访问限定符默认为`public`，因此`static void main()`与Java代码`public static void main()`等效。
- 使用`def`来声明变量时，无需在赋值语句的左侧指定类型。
- `println`是Groovy开发包中的一个内置函数，相比于`System.out.println`，它更紧凑。
- Groovy甚至不要求你使用括号（`()`）将方法名和传入的值分开。只要编译器确定不存在二义性，就会允许你这样做。在前述代码中，`println p.name`完全合法，编译器不会提出异议。
- 可直接通过属性名来访问属性。在前面的示例中，Groovy将分别调用方法`setName()`和`getName()`。

在本书后面，将更详细地介绍前述众多要点。

虽然在Groovy中括号并非必不可少，但这并不意味着在任何情况下省略它们都是好主意。很多程序员发现，通过用括号将方法名和输入参数分开，可提高代码的可读性。



Groovy官方风格指南推荐在特定的情况下省略括号。本书不介绍Groovy风格指南，但你可参阅http://groovy-lang.org/style-guide.html。

Groovy 面向对象编程

在面向对象编程方面，下面这些无疑是Java和Groovy最大的不同之处。

- 不同于Java，Groovy是一种完全面向对象编程的语言。
- 在Groovy中，默认访问限定符为`public`。
- Groovy能够自动为属性创建获取和设置函数。
- 对于属性、变量以及方法的参数和返回值，可显式地指定它们的类型，也可不指定。
- Groovy能够自动创建功能齐备的POJO。
- 可自动生成不可变类。

1. Groovy是完全面向对象的

相比于Java，Groovy的一个不同之处在于它是完全面向对象的：它不创建基本类型值，而是在任何情况下都创建对象。为验证这一点，请在GroovyConsole（或GroovyShell）中运行下面的脚本：

```
int i = 555
i.getClass()
```

这将打印`java.lang.Integer`（这可能让你感到意外）。在Java中，这将创建一个`int`值，进而拒绝编译上述代码，但Groovy在任何情况下都创建对象。然而，Groovy完全与Java工具和库兼容，因为它将基本类型值自动装箱为包装类，反之亦然。

2. 访问限定符

在程序员没有显式给方法或类/实例变量指定访问限定符时，Java认为它是包私有的，而Groovy默认使用访问限定符`public`。



这是一个重要的差别。有鉴于此，大部分Java代码虽然在语法层面与Groovy兼容，但最终的行为可能不同，并可能以无法预料的方式修改程序。

Groovy支持如下访问限定符：

- `public`；
- `protected`；
- `private`。

在没有显式地指定访问限定符时，Groovy默认使用`public`；同时Groovy没有提供与Java包私有对应的关键字。有鉴于此，Groovy本身不支持创建包私有的类或成员。



为应对必须使用包私有成员的情形（这种情形很少见），Groovy开发包（GDK）提供了注解`@PackageScope`，你可从`groovy.transform`包中导入它。然而，本书不会介绍这个高级主题。

必须指出的是，在类的私有成员方面，Groovy存在一个由来已久的bug：不遵守有关访问限

定符private的约定。下面的Java代码无法通过编译（在任何遵守有关访问限定符private的约定的语言中，类似的代码都无法通过编译），但在当前的Groovy版本中却能畅通无阻地运行：

```
public class MainDemoClass {
  public static void main(String[] args) {
    ClassWithSecret secret = new ClassWithSecret();
    System.out.println(secret.privateVariable);
  }
}

class ClassWithSecret {
  private int privateVariable = -1;
}
```

实例变量privateVariable使用了访问限定符private，在其所属的类ClassWithSecret外面应该是不可见的。在生成的Java字节码中，Groovy编译器正确地将privateVariable设置成了private的，因此在Java等语言中使用这个类时，将无法在类似于上述的代码中访问privateVariable（除非采用访问私有数据的低级技巧）。



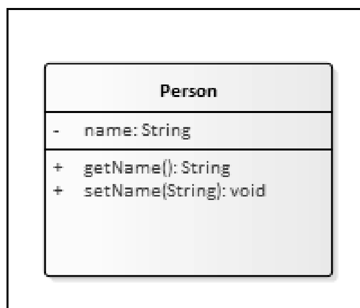
在这方面，Groovy的行为类似于Python等动态语言。Python其实并没有公有和私有成员的概念——一切都是公有的。多年来，对于Groovy不支持限定符private是bug还是特色一直争论不休。

3. 给类添加属性

对于没有显式地指定访问限定符（public、private或protected）的属性，Groovy编译器将自动为它创建一个同名的私有变量，并生成公有的设置函数和获取函数：

```
class Person {
  String name
}
```

上述代码创建的类如下：



Groovy编译这些代码时，将完成如下工作。

- 创建私有变量name。
- 创建类似于public void String getName()的公有获取函数。
- 创建类似于public void setName(String name)的公有设置函数。

仅当没有指定访问限定符时，编译器才会这样做。只要指定了访问限定符（无论是private、public还是protected），编译器就认为开发人员想全面控制属性，进而只创建一个变量，让开发人员根据需要决定是否创建获取函数和/或设置函数。

与Kotlin一样，Groovy也不要求通过调用获取函数和设置函数来访问属性，只需使用属性名即可。请在前面的示例中添加如下代码，再运行它：

```
p = new Person()
p.name = "D. Vader"
println(p.name)
```

这些代码调用Groovy自动创建的方法getName()和setName()。为证明这些代码没有利用前面提及的访问限定符private存在的bug，请将Person类的代码替换为如下代码：

```
class Person {
  private String personName;
  public void setName(String name) { this.personName = name }
  public String getName() { return this.personName }
}
```

如果你尝试运行这些代码，将发现它们能够畅通无阻地运行而且管用，虽然现在没有私有变量name。Groovy确实调用了setName()和getName()。

4. 类型是可选的

Groovy全面支持Java的声明风格。在Java中，同时指定变量类型和使用的实例类型，如下所示：

```
Date date = new Date();
```

在Groovy中，声明变量时可不指定其类型，为此可使用关键字def：

```
def date = new Date()
```

这类似于Java语句Object date = new Date()；差别在于在Groovy中使用关键字def声明变量date时，通过这个变量可访问java.util.Date类的所有成员：

```
def date = new Date()
println date.getTime()
```


在Java中，必须将Object实例向下转换为java.util.Date实例，才能访问java.util.Date的成员：

```
// Java代码（假定导入了java.util.Date）
Object date = new Date();
System.out.println(((java.util.Date)date).getTime());
```

如你所见，Groovy代码不但简洁得多，可读性也更强。在这方面，Java和Groovy的主要差别在于，Java会在编译阶段检查方法是否可用。Java编译器检查在变量的引用类型中是否有指定的方法。因此，只有将这个变量向下转换为`java.util.Date`类后，它才会接受方法`getTime()`，因为`java.lang.Object`类没有方法`getTime()`。而Groovy更灵活，它等到运行阶段才尝试调用方法，并在对象不支持调用的方法和/或传入的参数时引发异常。Groovy不关心引用变量的类型。

需要指出的是，使用`def`声明的变量可指向任何类型的对象：

```
def d = new Date()
d = new ArrayList()
```

对于方法的参数和返回值，也可不指定类型：

```
def methodWithParameters(parm1, parm2, parm3) {
  // 代码……
}
```

对于这里的参数`parm1`、`parm2`和`parm3`以及返回值，类型为`java.lang.Object`。

另外，`return`语句也是可选的。函数的返回值默认为其最后一个表达式：

```
int methodWithImplicitReturnValue(int i) {
  i * 10
}
```

最后，定义属性时，也可不显式地指定其类型：

```
class Sensor {
  def temperature
}
```

在这个示例中，将创建类型为`java.lang.Object`的私有变量`temperature`，还将创建获取函数`public Object getTemperature()`以及设置函数`public void setTemperature(Object temperature)`。



在大多数情况下，最好还是指定类型，否则小组的其他开发人员必须阅读注释来确定哪些类型与你创建的方法和属性兼容。可你在任何情况下都会给代码添加注释吗？

5. 自动创建功能齐备的POJO

如果在类名前指定了注解`@Canonical`，Groovy将生成如下元素——如果类本身没有这些元

素的自定义实现的话。

- ❑ 不接受任何参数的构造函数。
- ❑ 将每个属性都作为参数的构造函数（这些参数的排列顺序与属性的定义顺序相同）。
- ❑ 方法`toString()`的实现，它打印所有的属性（名称和值）。
- ❑ 方法`hashCode()`的实现。
- ❑ 方法`equals()`的实现。

如果类实现了`toString()`、`hashCode()`或`equals()`，Groovy将不考虑它们，继续检查其他元素是否实现了。因此，如果你在自定义类中实现了`toString()`，Groovy将使用该实现：

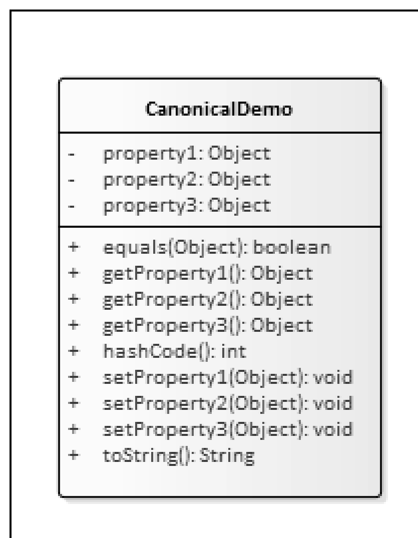
```
import groovy.transform.Canonical
@Canonical
class CanonicalDemo {
  def property1
  def property2
  def property3
}

def demo = new CanonicalDemo("value for property1", "value for
property2")
println("${demo.property1}, ${demo.property2}, ${demo.property3}")
println(demo)
```



这里演示了Groovy字符串的一种强大功能：它们提供了内置的模板支持，这将在后面更详细地讨论。

下图概述了为`CanonicalDemo`类生成的所有属性和方法。



前面的代码将打印value for property1, value for property2, null和CanonicalDemo (value for property1, value for property2, null)。注意，为Canoni calDemo类生成的方法toString()只按声明顺序打印属性的值，而没有打印属性的名称。

正如刚才演示的，并非必须给所有的属性都提供值。另外，也可只给特定的属性指定值：

```
def demo = new CanonicalDemo(property1:"value 1", property3: "value3")
```

如果你只想让Groovy生成方法hashCode()和equals()的实现，而不想让它生成方法toString()和构造函数，可使用注解@EqualsAndHashCode。同样，如果你只想让Groovy生成方法toString()的实现，可使用注解@ToString。最后，如果你只想让Groovy生成构造函数，可使用注解@TupleConstructor。在这些注解中，有些有可选的参数，让你能够实现更细致的控制。本书不会对这个主题做更深入的讨论，详情请参阅文档。



前述所有注解都必须从groovy.transform包中导入后才能使用。

6. 创建不可变类

正如你在前几章看到的，不可变类是函数式编程的基石。Groovy提供了很多可帮助你以函数式编程风格编写代码的功能。众所周知，可变类是bug的温床，因此即便你不打算使用Groovy进行大量的函数式编程，最好也应将这些类声明为不可变的。要将类声明为不可变的，可使用groovy.transform包中的注解@Immutable：

```
import groovy.transform.Immutable

@Immutable
class Person {
  String name
}
```

与注解@Canonical类似，这个注解也让编译器生成如下元素：

- 不接受任何参数的构造函数；
- 将每个属性都作为参数的构造函数；
- 方法hashCode()的实现；
- 方法equals()的实现；
- 方法toString()的实现。

除让编译器生成上述元素外，@Immutable还：

- 将类声明为final的，使其无法被其他类继承。
- 检查所有属性的类型，看它们是否是不可变的。如果有属性不是不可变的或根本不支持不变性，@Immutable将引发异常。

❑ 确保所有的设置方法都将在必要时引发异常。

如果使用@Immutable注解的类包含一个或多个类型为自定义类的属性，这些自定义类也必须使用@Immutable注解；否则Groovy将拒绝编译这个类。我们来看一个示例：

```
import groovy.transform.Immutable

class Person {
  public final String name
  public Person(String name) { this.name = name }
}

@Immutable
class Demo {
  // 不能运行
  Person person = new Person("test")
}

def d = new Demo()
```

Groovy将拒绝运行上述代码，因为它不知道Person类是否是不可变的：

```
java.lang.RuntimeException: @Immutable processor doesn't know how to handle
field 'person' of type 'Person' while constructing class Demo.
```

要让这些代码能够运行，请将@Immutable替换为如下代码行：

```
@Immutable(knownImmutableClasses=[Person])

class Demo {
  ...
}
```

这样，Groovy将相信你的直觉，认为将这个类标记为不变的是安全的。还可指定属性名（而不是类型）：

```
@Immutable(knownImmutables=["person"])
```

这在你想使用def person = new Person()（而不是Person person = new Person()）时很有用。



当然，如果直接给Person类本身添加注解@Immutable，代码将更容易理解和维护。这样做后，Groovy将知道Person类是不变的，因此你无需手动调整Demo类。

11.3　Groovy 开发包（GDK）

Groovy自带了一个庞大的类库，你可使用它来简化工作。在这个类库中，有些类提供了新功能，而有些是Java类库中类的包装器，旨在让Java类使用起来更容易或改进它们的功能。本节介

绍Groovy运行时库中的一些重要类和类型。Groovy运行时库是随Groovy一起安装的，有时也被称为Groovy开发包（GDK）。

11.3.1　Groovy 字符串（GString）

Groovy提供了java.lang.String类的变种——groovy.lang.GString。每当你使用双引号创建字符串时，Groovy都会检查你是否使用了GString的功能。如果使用了，它就创建GString实例，否则创建java.lang.String实例：

```
def s = "this is an ordinary java.lang.String instance";
```

这个字符串没有使用GString的功能，因此Groovy创建一个java.lang.String。GString最有用的功能之一是提供了内置的模板支持：

```
def who = "you"
def msg = "Happy birthday to $who"
```

运行上述代码时，msg将包含Happy birthday to you。由于在字符串中使用了变量，因此上述代码将创建GString。使用模板变量时，必须使用大括号来消除二义性：

```
def who2 = "packtpub"
def msg2 = "Please visit ${who2}.com"
```

如果省略大括号，上述代码将崩溃，因为who2指向一个java.lang.String实例，而这种实例没有属性com。

要在字符串中使用美元符号，一种办法是使用反斜杠对其进行转义："US\$ 100"表示字符串US$100。另一种选择是创建Java字符串，在Groovy，这可使用单引号来实现：

```
def javaString = 'This is a Java string, even though it has ${who}'
```

上述代码将创建一个java.lang.String，同时不会将${variable}替换为相应的变量。这与Java不同；Java用单引号来表示char字面量值。

当你将类型声明为char（或其包装类java.lang.Character）时，Groovy将创建java.lang.Character实例（别忘了，Groovy在任何情况下都不会创建基本类型值）：

```
char c = 'C'
c.class
```

上述代码片段的结果为java.lang.Character。请注意，代码char c = "C"（使用双引号）也将创建一个java.lang.Character实例，但在Java语言中，这样的代码会导致编译错误。

最后，与众多其他的现代语言一样，Groovy也支持多行字符串（GString和Java字符串都如此）：

```
def longMsg = """
  Happy birthday
  to ${who}
"""
```

与单行的GString实例一样，多行的GString实例也支持变量。要创建多行的Java字符串，可使用三个单引号：

```
def longJavaMsg = '''
  Another long
  message
'''
```

最后，需要指出的是，在Groovy中可使用运算符==和!=对字符串进行比较。在这种情况下，Groovy将调用方法equals()来比较字符串的内容：

```
def s1 = "hello"
def s2 = 'hello'
println(s1 == s2)
```

在这个示例中，两个字符串都是java.lang.String实例（因为没有使用GString的功能）。如果运行这些代码，将向控制台打印true。即便s1是一个包含"hello"的GString的实例，也将打印true。

11.3.2　集合

在集合方面，Groovy将类似于Python的体验带到了JVM中。对于Java类库中最重要的集合，它提供了内置的支持，同时给它们添加了额外的功能。本节将介绍如下主题：

- 列表；
- 映射。



与Java相比，最大的不同在于Groovy集合不支持泛型。虽然Groovy解析器支持Java泛型语法，但Groovy不会以任何方式实现泛型。

1. 列表

要创建列表（java.util.ArrayList实例），只需使用中括号即可：

```
def list = [10, 20, 30, 40, 50]
```

要获取列表中的元素，可使用中括号并在其中指定索引：

```
println list[1]
```

这将返回20。在这个示例中，这行代码与list.get(1)完全等价。但存在一个差别：如果指定的索引不在有效范围内，使用中括号时结果将为null，而使用方法get()时将引发

IndexOutOfBoundsException异常。

使用中括号时，可指定负索引以倒数的方式获取元素：-1表示最后一个元素，-2表示倒数第二个元素，以此类推。请看下面的示例：

```
println list[-4]
```

这行代码返回20。请注意，在方法get()中不能指定负索引，否则将引发IndexOutOfBoundsException异常。

要对列表执行切片操作，可使用Groovy的下标运算符..（两个句点）：

```
println list[1..2]
```

指定的两个索引都包含在切片内，因此这将返回[20, 30]。

除指定范围外，还可指定一系列索引：

```
println list[0,3]
```

这将返回[10, 40]。另外，还可同时指定范围和特定的索引：

```
println list[0..2, 4, 3]
```

这将返回[10, 20, 30, 50, 40]。

在真正的动态编程中，空列表对应false，而包含元素的列表对应true。可利用这一点来检查列表是否是空的：

```
def emptyList = []
if (!emptyList) {
  println("List is not empty")
}
```

上述代码不会打印任何输出。在Java 8推出之前很久，Groovy就给Java集合类添加了函数式编程功能。要遍历集合，可指定一个闭包：

```
list.each({
  def bar = "X" * it
  println "${bar} ${it}"
})
```

将对每个元素调用这个闭包，而变量it包含当前元素的值。在这个闭包中，创建了变量bar，它包含将X重复n次的结果（其中n为当前元素的值）。上述代码的输出如下：

```
XXXXXXXXXX 10
XXXXXXXXXXXXXXXXXXXX 20
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 30
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 40
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 50
```

包括java.util.ArrayList在内的大多数集合类都继承了接口java.util.Collection，而在这个接口（及其继承的接口）中，Groovy添加了一些非常方便的方法，下表列出了其中的几个。

方 法 名	描 述	示 例
any(Closure)	如果至少有一个列表元素导致闭包返回true，这个方法就返回true	list.any { it > 20 } 返回true
every(Closure)	如果所有的列表元素都导致闭包返回true，这个方法就返回true，否则返回false	list.every { it < 50 } 返回false
find(Closure)	对每个元素调用闭包；如果闭包返回true，将停止迭代。如果找到指定的元素，就返回它；否则返回null	list.find { it == 30} 返回30
findAll(Closure)	类似于find()，但不在闭包返回true时停止迭代	list.findAll { it > 30} 返回[40, 50]
join(String)	将所有元素合并成一个字符串，并用指定的字符分隔元素	list.join("/") 返回10/20/30/40/50
min()	返回最小的元素	list.min() 返回10
max()	返回最大的元素	list.max() 返回50
sum()	返回所有元素之和	list.sum() 返回150

2. 映射

要创建映射，可使用中括号并在其中指定键-值对：

```
def map = [ key1: "value1", "key2": "value2" ]
```

键的默认类型为字符串，因此指定键时，可以不将其用引号括起。在需要添加基于变量的键或类型不是字符串的键时，这可能是个问题；在这些情况下，必须使用括号将键括起：

```
def key1 = "whateverKey"
def otherMap = [ (key1): "whateverValue"]
```

要创建空的映射，必须使用如下表示法：

```
def emptyMap = [:]
```

要读写映射，可使用中括号：

```
map["key1"] = "anotherValue1"
println map["key1"]
```

Groovy将映射视为POJO，因此可使用句点表示法来读写映射：

```
map.key1 = "yetAnotherValue1"
println(map.key1)
```

仅当键为字符串（合法的Java标识符）时，这种表示法才可行。如果键不是字符串，就必须使用中括号或方法get：

```
map[30] = "thirty"
println(map.get(30))
```

由于30不是字符串（不是合法的Java标识符），因此必须使用映射的方法get()或中括号表示法来访问它。

由于接口Map继承了接口Collection，因此映射支持前面介绍的所有方法。不同之处在于，闭包将一个MapEntry对象作为参数，并通过其属性key和value来获取每个元素的键和值。

来看一些遍历键–值对的示例：

```
map.each({
  println("$it.key --> $it.value")
})
```

在闭包中也可使用键–值对。在这种情况下，必须给闭包指定两个参数：

```
map.find({ k, v -> k =="key2" && v == "value2" })
```

在这个示例中，参数k和v分别表示映射中每个元素的键和值。

11.4　动态和静态编程

静态类型编程语言和动态类型编程语言的一个重要差别在于，对于静态编程语言，编译得到的程序（就JVM语言而言，为生成的Java字节码）包含的方法调用和引用是经过解析或编译的，而Groovy、Clojure和Python等动态编程语言等到程序运行阶段才做出这方面的决策。

与人生的其他方面一样，这两种方法都有优点和缺点。

编程风格	优　　点	缺　　点
静态	❑ 应用程序的运行速度快 ❑ 能够在编译阶段发现很多细微的错误 ❑ 顶级IDE提供了相应的支持，它们提供了卓越的重构工具，可极大地提高效率	❑ 编译阶段通常耗时很长 ❑ 为满足编译器的要求，通常需要编写更多的代码
动态	❑ 通常需要编写的代码更少 ❑ 编译速度通常非常快 ❑ 支持元编程	❑ 应用程序的性能通常较低 ❑ 很多错误要到运行阶段才能发现；细微的错误可能隐藏很长时间 ❑ 需要编写代码来验证传递给函数的参数类型 ❑ IDE提供的支持有限，因为提供相关的支持要难得多

在动态语言（如Groovy和Clojure等JVM语言以及JavaScript、Python和Ruby等非JVM语言）中调用方法时，要等到运行阶段才判断对象实例是否有相应的方法，如果有，再判断传入的参数是否合法。而在静态语言（如Java、Scala和Kotlin等JVM语言以及C、C#、C++和Visual Basic等非JVM语言）中，这些工作是由编译器完成的，运行阶段只需直接执行编译后的指令。

显然，在运行阶段解析方法调用及访问属性需要占用一定的处理时间，因此你可能会问，这样做有什么好处呢？答案是能够进行元编程，且实现起来很容易，这将在下一节介绍。

11.4.1　元编程

为了理解元编程的概念，我们来创建一个小型类：

```
class MetaProgrammingDemo {
}

def demo = new MetaProgrammingDemo()
// 接下来的代码会引发异常!
demo.nonExistingProperty = "some value"
println(demo.nonExistingProperty)
```

如你所料，当JVM执行最后一行代码时，将引发异常：

```
groovy.lang.MissingPropertyException: No such property: nonExistingProperty
for class: MetaProgrammingDemo
```

接下来给MetaProgrammingDemo类添加如下方法：

```
def propertyMissing(String name) {
  println("Non-existent property '$name' was read")
  return -1
}

def propertyMissing(String name, args) {
  println("Non-existent property '$name' was written to: '$args'")
}
```

现在运行这些代码，JVM不会引发异常，而是向控制台打印如下输出：

```
Non-existent property 'nonExistingProperty' was written to: 'some value'
Non-existent property 'nonExistingProperty' was read
-1
```

注意，读取的属性值与代码设置的属性值不同，这是因为我们以硬编码的方式返回了-1。读取不存在的属性时，将调用方法propertyMissing(String name)，而设置不存在的属性时，将调用方法propertyMissing(String name, args)。

在Groovy中访问属性（无论是读取还是写入）时，Groovy将在运行阶段执行如下操作。

(1) 如果存在相应的获取/设置函数，就调用它。

(2) 如果不存在相应的获取/设置函数，就查找相应的实例变量或类变量；如果找到，就访问它。
(3) 如果没有相应的变量，就查找方法`propertyMissing`；如果找到，就调用它。
(4) 如果上述操作都以失败告终，就引发`groovy.lang.MissingPropertyException`异常。



实际流程比上面描述的更复杂，因为除本章讨论的元编程方式外，Groovy还提供了其他元编程方式。

在动态编程语言中，可让代码以为原本不存在的属性是存在的，这在静态类型语言中几乎是无法实现的。这种技术为何很有用呢？在什么情况下很有用呢？目前这些对你来说可能不那么显而易见。下一章将使用Groovy的XML生成器，它大量地使用了元编程，可能会让你对如何在自定义类中使用这种强大的技术有一定的了解。

前面只处理了不存在的属性；对于方法，也可使用类似的技术。请在`MetaProgrammingDemo`类中添加如下方法：

```
def methodMissing(String name, args) {
  println("Non-existent method '$name' was called with '$args'
          parameters")
}
```

再在既有的测试代码中添加如下代码：

```
demo.methodThatDoesNotExist(1000, "demo")
```

现在如果运行这些代码，将向控制台打印下述额外输出：

```
Non-existent method 'methodThatDoesNotExist' was called with '[1000, demo]'
parameters
```



这种技巧仅适用于Groovy。如果你在其他JVM语言中使用这个类，JVM不会在遇到读写未知属性的语句时自动调用方法`missingProperty()`，也不会在遇到未知的方法调用时自动调用`missingMethod()`。

11.4.2　Groovy 静态编程

要让编译器在编译特定的类或方法时切换到静态编程模式，可使用注解`@TypeChecked`，但在此之前必须将其从`groovy.transform`包导入：

```
import groovy.transform.TypeChecked

@TypeChecked
class TypeCheckedClass {
}
```

如果你给类指定了这个注解，编译器将像静态类型语言的编译器那样做大量的检查，并将直接引用编译成方法和属性，而不再让代码在运行阶段再决定必须调用哪个方法。这可让代码的执行速度更快些，但实际上，仅当代码在循环中被多次调用或被多个线程同时调用时，这种差别才能显现出来。

使用注解`@TypeChecked`后，类就不能使用元编程等动态编程功能。例如，下面的代码无法通过编译：

```
import groovy.transform.TypeChecked

@TypeChecked
class Demo {
  static void main(String[] args) {
    def d = new Demo()
    // 无法通过编译
    d.thisMethodDoesNotExist()
  }

    def methodMissing(String name, args) {
      println("Method '$name' was called")
  }
}
```

如果在GroovyConsole中运行这些代码（运行代码前，GroovyConsole也会在幕后对其进行编译），将出现如下编译错误：

```
[Static type checking] - Cannot find matching method
Demo#thisMethodDoesNotExist(). Please check if the declared type is right
and if the method exists.
```

鉴于编译器不确定`methodMissing()`会不会支持调用`thisMethodDoesNotExist()`（它可能引发异常，也可能只处理特定的方法名），因此拒绝对这些代码进行编译。要解决这个问题，可将注解`@TypeChecked`从`Demo`类中删除。但还有另一种解决方案：让编译器的类型检查器忽略某些方法。为此，可在代码行`static void main(String[] args)`前面添加如下注解：

```
@TypeChecked(groovy.transform.TypeCheckingMode.SKIP)
static void main(String[] args) {
  ...
}
```

这让类型检查器不要检查方法`main()`。当然，如果你在某个类的很多方法前面都添加了这个注解，也许根本就不需要给这个类添加注解`@TypeChecked`。注解`@TypeChecked`也可用于方法：


```
class Demo2 {
  def static void main(String[] args) {
    def d = new Demo()
    d.typeCheckedDemoMethod()
```

```
    }

    @TypeChecked
    def typeCheckedDemoMethod() {
      // 静态类型实现代码……
    }
}
```

11.5　小测验

(1) Groovy在很大程度上与Java兼容。这是否意味着对于兼容的Java代码，Groovy编译它们的方式与Java编译器完全相同，没有任何副作用呢？

a) 是的，Java编译器和Groovy编译器生成的Java字节码完全相同，编译得到的类的行为也完全相同。
b) 不对，在语法层面，Java和Groovy不完全兼容。
c) 不对，遇到Java基本数据类型时，Groovy将崩溃。
d) 不对，在语法层面，Groovy在很大程度上与Java兼容，但由于Groovy开发小组在设计方面做出的选择不同，对于同一个类，使用Groovy编译器和Java编译器编译后，行为可能不完全相同。

(2) 在下面的代码中，变量`msg1`的值是什么？

```
def name1 = 'reader'
def msg1 = "hello, $name1"
```

a) `"hello, reader"`。
b) `"hello, $name"`。
c) 这个程序会引发异常。
d) 以上答案都不对。

(3) 在下面的代码中，变量`msg2`的值是什么？

```
def name2 = "reader"
def msg2 = 'hello, $name2'
```

a) `"hello, reader"`。
b) `"hello, $name"`。
c) 这个程序会引发异常。
d) 以上答案都不对。

(4) 在下面的代码中，变量`longValue`是哪种数据类型？

```
long longValue = 999
```

a) 基本类型`long`。
b) `groovy.lang.Long`。
c) `java.lang.Long`。
d) 以上答案都不对。

(5) 判断对错：动态语言的一个优点是，生成的代码的运行速度总是比使用静态语言编写的等效代码的运行速度快。

a) 对。
b) 错。

11.6　小结

本章简要地介绍了动态的JVM语言Groovy。我们下载并安装了Groovy，同时探讨了它自带的两个REPL：GroovyConsole（桌面GUI应用程序）和GroovyShell（基于文本的shell）。我们发现，Java和Groovy语法在很大程度上兼容，但Groovy代码要紧凑得多，这是因为在Groovy中，很多在Java中必不可少的元素都是可选的。我们尝试使用了各种自动生成代码的注解，包括为方法`toString()`、`equals()`和`hashCode()`生成有效实现并生成完整构造函数的注解。我们粗略地探索了Groovy开发包（GDK），研究了动态编程和静态编程的差别，并发现Groovy对这两种方法都提供了支持。

下一章将创建一个简单的Web服务，它生成XML并通过接口Java Database Connectivity（JDBC）来使用数据库中的数据。在这个过程中，我们将使用刚学到的Groovy知识，还将探索GDK中的其他类。

第 12 章 Groovy编程

本章将使用流行的微服务框架Vert.x创建一个简单的Groovy Web服务，它使用了完全使用Java编写的H2数据库管理系统（DBMS）。我们将使用Java Database Connectivity（JDBC）标准来与H2交互。XML将使用Groovy的MarkupBuilder来生成，MarkupBuilder是Groovy运行时库，即Groovy开发包（GDK）提供的一个类。

这次我们不使用外部构建工具来构建项目，而是让Eclipse IDE来替我们处理这项工作。为了使用前面提到的开源项目H2和Vert.x，需要下载一些外部依赖项，我们将使用Apache Ivy来完成这项任务。Eclipse IDE本身不支持Groovy，因此我们需要安装一个支持这种语言的插件。本章介绍如下主题：

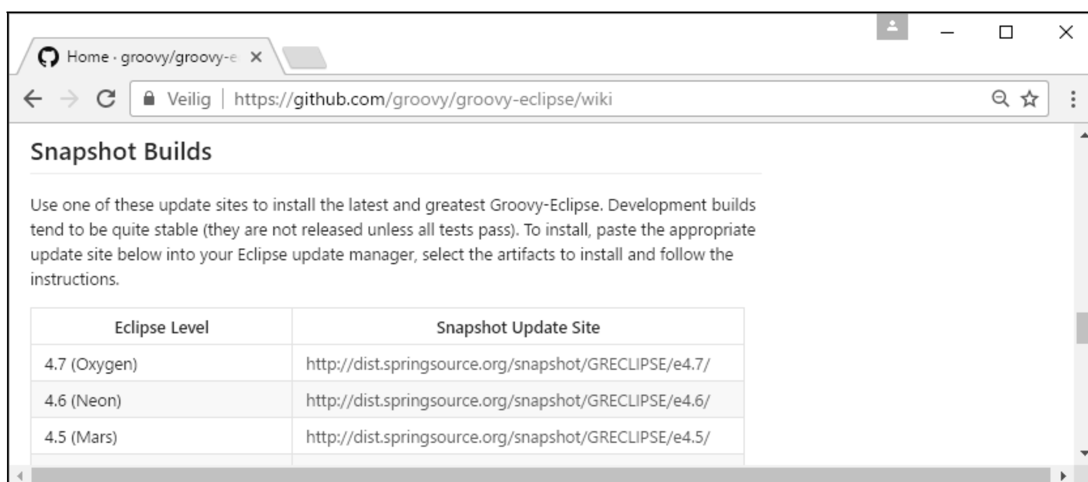
- 安装Groovy Eclipse插件；
- Apache Ivy和Eclipse IvyDE插件；
- 创建并配置Groovy项目；
- Java Database Connectivity（JDBC）;
- 使用MarkupBuilder生成XML；
- 微服务平台Vert.x。

12.1 安装 Groovy Eclipse 插件

让Eclipse IDE支持Groovy的插件Groovy Eclipse可在Eclipse Marketplace中找到，但在编写本书期间，Eclipse Marketplace提供的版本不是最新的。有鉴于此，我们将手动从Groovy Eclipse开发小组的服务器下载并安装它。为此，请访问该项目的GitHub页面（https://github.com/groovy/groovy-eclipse/wiki），找到正确的下载链接。

在Eclipse IDE中选择菜单Help>About，以确定你当前使用的Eclipse IDE版本。然后，在前述GitHub页面中，向下滚动到Releases部分并查找发行版本。如果没有用于你当前使用的Eclipse IDE版本的稳定版本，请找到Snapshot Builds部分，并在其中查找你当前安装的Eclipse IDE版本。编写本书期间，我安装的是Eclipse Neon（4.6），但没有用于该版本的Groovy Eclipse稳定版本，因

此不得不使用snapshot build：

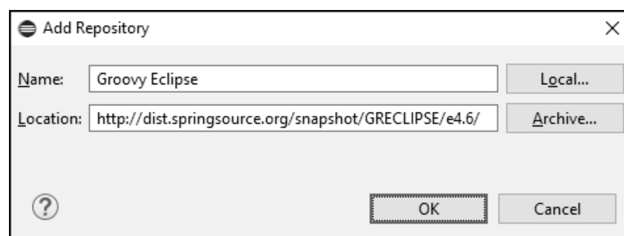




如果你安装的Eclipse IDE版本较新，没有用于它的稳定的Groovy Eclipse发行版，或者开发版不够稳定，你可下载较旧的Eclipse版本，并使用它来进行Groovy开发。

要在Eclipse IDE中安装你选择的Groovy Eclipse版本，请执行如下步骤。

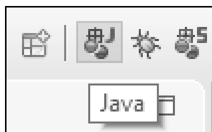
(1) 将GitHub页面中该版本的发行更新网站（Release Update Site）URL复制到剪贴板。
(2) 在Eclipse IDE中，选择菜单Help>Install New Software...。
(3) 在对话框Available Software中，单击文本框Work with旁边的按钮Add。
(4) 在文本框Name中输入Groovy Eclipse，将前面复制的URL粘贴到文本框Location中，再单击OK按钮：



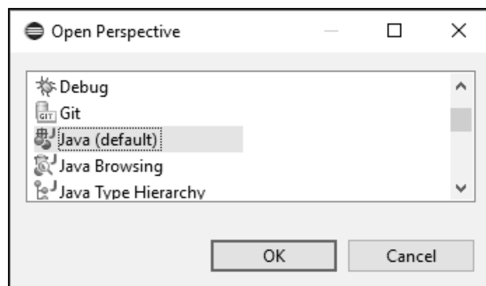
(5) 对话框Available Software将显示找到的包。选中复选框Groovy-Eclipse（required），再单击Next按钮。
(6) 可能出现一个对话框，指出修改了你所做的选择——添加了Groovy编译器。单击Next按钮确认这没问题。另外，接受许可条款，并单击Finish按钮安装这个插件。
(7) Eclipse IDE询问是否要立即重启时，选择Yes。

切换到 Java 透视图

Groovy Eclipse插件不会在Eclipse IDE用户界面中添加专用的Groovy透视图，而是像Clojure插件Counterclockwise一样，使用普通的Java透视图。请在屏幕右上角的工具栏中，单击Java透视图按钮：



也可单击这个工具栏中工具提示为“Open Perspective”的按钮，再选择Java(default)并单击OK按钮：



12.2　Apache Ivy 和 IvyDE

不像本书前面介绍的其他语言那样，这里不使用额外的构建工具来构建项目，而是使用Eclipse IDE内置的基于Apache Ant的构建功能。Ant是最先流行的JVM的构建工具。开发本章的项目时，我们将让Eclipse IDE负责完成构建过程。



很多流行（以及一些不那么流行）的JVM构建工具都支持Groovy。构建基于Groovy的项目时，如果你要更全面地控制IDE提供的构建过程，Gradle和Maven都是不错的选择。

为创建这个访问数据库的Web服务示例，需要使用多个外部依赖：

- 用于创建微服务的Vert.x框架；
- 一个本地数据库管理系统（DBMS），包括JDBC驱动程序。

我们可从各个网站手动下载所需的文件，将其安装到正确的目录并调整JVM类路径，但这需要做大量的工作，因为依赖项本身也可能依赖其他外部库。Groovy有一个内置的依赖管理器——

Grape，但在Groovy Eclipse中使用它时会出问题。

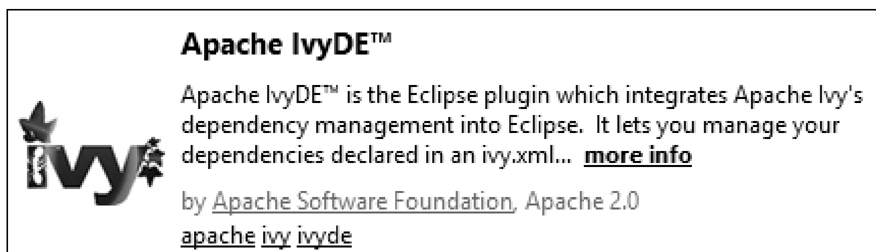
有鉴于此，本章将使用Apache Ivy来管理依赖。Ivy是一个依赖管理器（而不是构建工具），它与Maven仓库兼容，并知道托管仓库的最流行服务器。如果需要Ivy默认不支持的自定义服务器中的依赖项（本章的示例不需要），可轻松地添加相应的服务器定义。Ivy常与构建工具Apache Ant结合起来使用，因为Apache Ant本身没有提供依赖管理功能，但Ivy是一款完全独立的产品。

为了支持Ivy，需要在Eclipse IDE中安装一个插件Apache IvyDE。

安装用于 Eclipse IDE 的 Apache IvyDE 插件

要安装Apache IvyDE插件，请执行如下步骤。

(1) 选择菜单Help>Eclipse Marketplace...。
(2) 搜索Ivy。找到Apache Software Foundation出品的Apache IvyDE，并单击相应的Install按钮：



(3) 按提示操作。将出现一个警告对话框，指出代码未经签名；如果你要使用这个插件，就得接受这一点。最后，Eclipse会询问你是否要立即重启，请单击Yes。

安装Apache IvyDE插件后，就可以开始开发项目了。

12.3 创建并配置项目

在Eclipse IDE中安装所有必要的插件后，就可以创建项目了。我们还将定义并下载这个项目的第一个外部依赖。本节介绍如下主题：

- 新建Groovy Eclipse项目。
- 创建供Ivy使用的ivy.xml文件。

12.3.1 新建 Groovy Eclipse 项目

为了在Eclipse中创建基于Groovy的项目，请执行如下步骤。

(1) 右击Package Explorer的空白区域，并选择New>Other...。
(2) 在Select a wizard对话框中，选择Groovy>Groovy Project并单击Next按钮。
(3) 将项目名设置为GroovyWebservice。
(4) 单击Finish按钮生成项目。

Groovy Eclipse生成的项目只包含骨架，而没有任何示例文件，下面来创建一个以检查安装情况。

(1) 右击项目的目录src中的(default package)，并选择New>Other...。
(2) 在Select a wizard对话框中，选择Groovy>Groovy Class并单击Next按钮。
(3) 输入包名`webservice`和类名`Main`。
(4) 单击Finish按钮生成这个类。

同样，Groovy Eclipse创建的类非常简单：

```
package webservice

class Main {
}
```

下面来添加一个简单的`main()`方法，以核实Groovy Eclipse得到了妥善的配置，能够编译并运行项目。请在`Main`类中添加如下方法：

```
static void main(String[] args) {
  println("Project is running fine!")
}
```

按Ctrl + F11（在macOS中为cmd + F11）或找到并单击工具栏中的Run图标。

可能出现错误消息，指出引用的项目不存在，也可能运行的是前一个项目，而不是你期望的项目。这是因为Groovy Eclipse当前不会在初始化项目时自动创建**运行配置**（Run Configuration）。如果你遇到了这种问题，可这样修复：在Eclipse IDE中选择菜单Run>Run Configurations...；在配置列表中找到Groovy Script并右击，再选择New以生成配置；单击Run按钮关闭窗口。现在Eclipse IDE将把这个配置与项目相关联，而你将在控制台中看到打印的消息。

12.3.2　创建供 Ivy 使用的 ivy.xml 文件

我们需要创建一个包含依赖信息的简单XML文件。IvyDE插件将使用Apache Ivy下载依赖并将其添加到正确的类路径（ClassPath）中。为创建这个ivy.xml文件，请执行如下步骤。

(1) 右击项目名并选择New>Other...。
(2) 选择IvyDE>Ivy File并单击Next按钮。
(3) 单击Container旁边的Browse按钮，并选择项目GroovyWebservice。

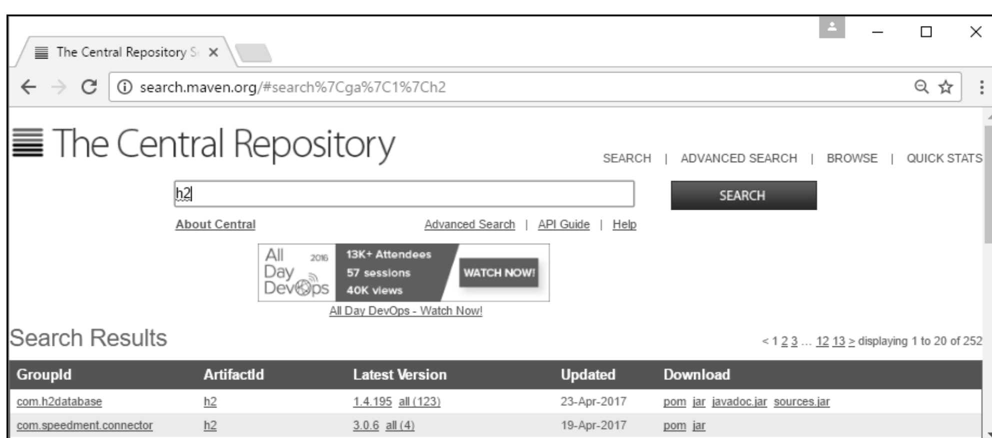
(4) 单击Finish按钮生成文件。

这将创建一个名为ivy.xml的文件，但其中没有包含任何依赖。下面来添加一个。

在这个示例中，我们将使用流行的基于文件的数据库系统H2。这里重点介绍如何为H2数据库系统下载必要的依赖，有关H2以及数据库连接的更详细信息将在下一节介绍。

我们在广泛使用的Maven中心仓库网站（The Central Repository）搜索H2数据库。为此，在你喜欢的Web浏览器中访问如下URL：http://search.maven.org。

在搜索栏中输入h2并按回车：



将出现一个列表，其中包含找到的依赖。请找到com.h2database（通常是第一个列表项），并单击Latest Version列的版本号——我看到的是1.4.195。

将出现一个列出了各种构建工具的页面。如果你阅读了本书前面的内容，将发现其中的一些名称是你很熟悉的，如Gradle、Scala SBT、Leiningen和Maven。为了下载必要的Ivy代码，请执行如下步骤。

(1) 单击Apache Ivy将其展开，以显示所需的Ivy XML条目。
(2) 将该XML条目复制到剪贴板。
(3) 在Eclipse中，打开前面生成的文件ivy.xml。
(4) 将前面复制的内容粘贴到标签`<ivy-module>`和`</ivy-module>`之间的最后面。

现在文件ivy.xml应类似于下面这样（为简洁起见，删除了大量的许可条件注释）：

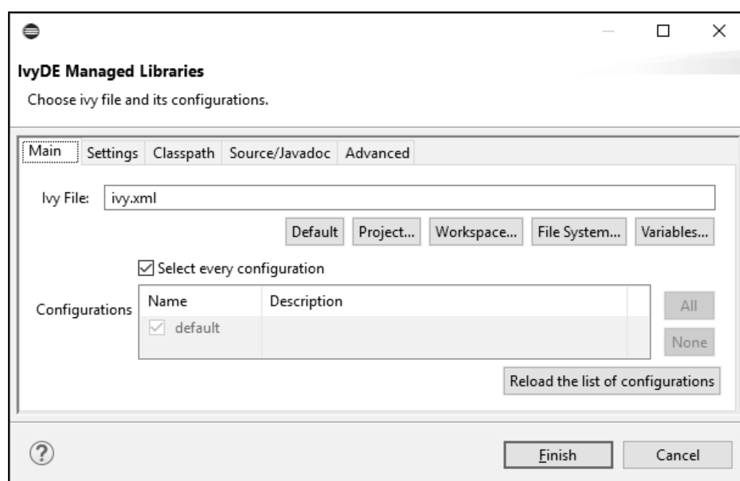
```
<ivy-module version="2.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://ant.apache.org/ivy/
                                schemas/ivy.xsd">
```

```
  <info
    organisation=""
    module=""
    status="integration">
  </info>
  <dependency org="com.h2database" name="h2" rev="1.4.194" />
</ivy-module>
```

你的版本号可能与这里列出的不同。

为了下载这些依赖并将其添加到类路径（ClassPath）中，请执行如下步骤。

(1) 右击文件ivy.xml并选择Add Ivy Library...：



(2) 在对话框IvyDE Managed Libraries中，单击Finish按钮。

Apache Ivy将从正确的仓库下载必要的H2 DBMS文件，并将它们都添加到项目的类路径（ClassPath）中。

12.4　Java Database Connectivity（JDBC）

Java Database Connectivity（JDBC）是一种标准，让你能够在JVM应用程序中访问**数据库管理系统**（DBMS）服务器。流行的企业级DBMS服务器包括：

- Oracle Database；
- Oracle MySQL；
- MariaDB；
- Microsoft SQL Server；

- IBM DB2；
- PostgreSQL。

要在JVM应用程序中使用JDBC连接到DBMS服务器，需要为目标数据库系统定制的JDBC驱动程序。应用程序将加载JDBC驱动程序，并提供一个通常包含服务器的主机名和端口以及凭证的连接字符串。JDBC系统将确保合适的驱动程序得以妥善地初始化，而该驱动程序将连接到数据库并返回一个`Connection`对象，供应用程序用来与数据库通信。



JDBC类似于Microsoft开发环境中的ADO.NET和ODBC标准。

JDBC标准并不要求DBMS服务器本身是使用Java或JVM实现的，但JDBC驱动程序通常是使用Java（或其他JVM语言）编写的。JDBC标准允许JDBC驱动程序使用原生（特定于平台的）驱动程序或库。使用原生软件的JDBC驱动程序安装起来可能更复杂，可能并非与所有JVM兼容平台都兼容。

JDBC驱动器分4类。

类　型	描　述
第1类：JDBC-ODBC bridge	在幕后使用基于ODBC的驱动程序与数据库服务器通信的JDBC驱动程序
第2类：原生API驱动程序	使用本地安装的原生驱动程序与数据库服务器通信的JDBC驱动程序
第3类：网络协议	连接到管理数据库连接的中间层（中间件）的JDBC驱动程序。中间件通常运行在独立的服务器上。与数据库通信的逻辑是在中间层实现的，因此客户端不需要针对特定数据库的驱动程序
第4类：数据库协议驱动程序	完全使用Java（或其他JVM语言）实现，因此独立于平台的JDBC驱动程序。这种驱动程序直接连接到数据库服务器

JDBC驱动程序通常可随JVM应用程序一起安装，为此只需将所需的文件放在JVM的应用程序类路径中即可。在有些情况下，驱动程序必须与JVM应用程序分开安装，这通常是因为需要随驱动程序安装平台特定的软件，或许可条款要求这样做。数据库厂商或开源小组负责为其产品开发和维护JDBC驱动程序。由于Java的市场影响力巨大，大多数流行的DBMS系统都有相应的JDBC驱动程序，Microsoft甚至免费提供用于其Microsoft SQL Server产品线的JDBC驱动程序。

创建、更新和删除记录（也被称为CRUD）的操作是使用流行的SQL查询语言来完成的。大多数数据库系统都支持通过SQL来新建数据库以及创建或更新表、索引和视图等。在很大程度上说，SQL查询语言已标准化，因此只要小心行事，编写的应用程序就能与各种不同的数据库服务器交互。由于每种数据库都有其独特的SQL语言扩展（包括定制的函数名、特殊的数据类型以及自定义查询语法），因此实际上前述目标很难实现。数据库系统可自由决定要支持哪些SQL语句和功能；JDBC标准对此并没有明确的规定。

12.4.1　H2 数据库

本章将使用H2数据库。H2是一种独立的开源DBMS系统，它相对较小，完全是使用Java编写的。这个数据库系统将文件写入本地文件系统，甚至将整个数据库都加载到内存中。它不像MySQL和PostgreSQL那样需要安装，同时它创建的数据库也不需要过多的维护。你只需将一些JAR文件添加到ClassPath中，JVM应用程序就能访问H2数据库系统及其属于第4类的JDBC驱动程序。



从某种程度上说，H2类似于流行的无版权（public domain）SQLite数据库。虽然有针对SQLite的开源包装器和JDBC驱动程序，但在Groovy等JVM语言中，H2使用起来更方便，因为编写它时全面考虑到了JVM。。

诸如H2等独立的数据库系统非常适合用于单用户应用程序，但对于需要向数百名同时读写数据库的用户提供服务，因此需要存储数GB数据或需要其他企业级可靠性或功能的多线程应用程序来说，这可能不是最佳的选择。然而，千万不要低估了H2这样的数据库系统的威力。例如，H2支持运行数据库服务器，这在多个应用程序需要访问同一个数据库时很方便；它还提供了高级集群选项。另外，它还自带了一个命令行工具，可用于通过方便的操作系统命令行界面来查询、分析、备份和恢复数据库。

建议你在阅读本章时随时参考H2数据库项目的主页（http://www.h2database.com）。

12.4.2　创建内存数据库

在这个项目中，我们将创建一个内存数据库，它在应用程序终止时将被立即删除。在原型阶段，这样做很方便，因为修改数据库结构后，手动更新数据库中的数据不会遇到任何麻烦。另一个优点是无需跟踪数据库存储路径。

要使用JDBC连接到数据库，必须提供一个连接字符串，它告诉JDBC应使用哪个驱动程序、数据库的位置以及访问数据库所需的凭证，它还包含随DBMS而异的配置选项。



在较旧的JDBC版本中，必须使用代码手动注册JDBC驱动程序。当前，较新的JDBC驱动程序会自动注册，因此通常只需将驱动程序的JAR文件放到应用程序的类路径中，它们就可供应用程序使用。

在这个示例中，我们将使用H2的嵌入模式，这意味着将整个H2数据库系统嵌入到应用程序中，因此无需运行外部服务器应用程序。我们将创建一个应用程序终止后就将消失的内存数据库。要访问这种H2数据库，必须向JDBC提供的连接字符串类似于下面这样：

```
String connectionString = "jdbc:h2:mem:blogs;DB_CLOSE_DELAY=-1"
```

连接字符串是一个用冒号分隔的列表，下面来看看其中的各个部分。

- JDBC连接字符串的第一项都是jdbc。
- 第二项是标识数据库系统的名称。JDBC驱动程序在加载期间注册其名称。由于Ivy已经将H2 JDBC驱动程序添加到项目的ClassPath中，因此JDBC系统能够识别名称h2。
- 第三项是数据库的名称。H2并不要求内存数据库有名称，但正如你将在下一项中看到的，在这个示例中实际上必须有名称。
- 第四项让H2将数据库保留在内存中，即便没有活动的连接亦如此。通常，对于没有活动连接的内存数据库，H2会将其删除，但我们希望只要应用程序在运行，就能访问这个数据库。为此，我们指定了选项DB_CLOSE_DELAY=-1。

在连接字符串中，只有前两项是标准的，其他项通常因JDBC驱动程序而异。

下面来编写一些代码。在Eclipse IDE中，打开文件Main.groovy。我们将创建一个打开新建数据库连接的方法，为此请在Main类中添加如下方法：

```
def createDatabaseConnection() {
  def connection = DriverManager.getConnection("jdbc:h2:mem:test;
                                               DB_CLOSE_DELAY=-1")
  return connection
}
```



请使用Eclipse IDE快捷键Ctrl +空格来补全类名DriverManager，这样Eclipse IDE将自动替你编写导入java.sql.DriverManager的语句。

变量connection包含的对象指向创建的数据库连接，可用来向数据库系统发布命令。变量connection所属类型的全限定名为java.sql.Connection，这是一个Java接口，但由于我们编写的是Groovy代码，因此无需指定类型。DriverManager是JDBC系统提供的一个类，知道如何访问已注册的JDBC驱动程序。

在这个示例中，我们使用的是嵌入的数据库系统，当H2的JDBC驱动程序将连接字符串传递给嵌入的H2数据库引擎时，H2将新建一个临时的内存数据库。应用程序终止时，这个数据库将被自动删除，因为我们在连接字符串中指定了选项DB_CLOSE_DELAY=-1。

我们在本章开头创建的是基于控制台的应用程序，但后面将修改其实现，使其变成Web服务。因此，下面来重写方法main()，使其调用刚创建的方法createDatabaseConnection()：

```
static void main(String[] args) {
  def app = new Main()
  def connection = app.createDatabaseConnection()
  connection.close()
}
```

方法main()是静态的，不能直接访问Main类的实例变量和实例方法。有鉴于此，我们创建Main类的一个实例，并使用这个实例来调用Main类的实例方法。

这里必须指出一个常见问题。必须将可打开的JDBC对象关闭，以防泄露宝贵的系统资源。打开数据库连接后，即便引发了异常，连接依然将处于打开状态，直到你将其关闭。这可能导致无法预料的问题和程序崩溃，因为数据库资源是有限的。因此，推荐使用`try...catch`块来使用JDBC对象，这样可在`finally`块中将连接关闭，这在第4章介绍过。通过这样做，可确保连接始终得以妥善地关闭，即便引发了异常。

现在可以创建数据库了，但空的数据库不是很有趣。我们需要创建一些用于存储数据的表，这些数据在关系型数据库中被称为记录。我们先直接使用JVM的JDBC类，然后再换成Groovy内置的JDBC包装类，这旨在让你明白使用Groovy与关系型数据库系统通信更容易。

我们将创建一个微型博客应用程序。首先来创建一个用于存储应用程序用户的表，为此在`Main`类中添加如下方法：

```
def createDatabaseStructure(connection) {
  def statement = connection.createStatement()
  def sqlUsers = """
    CREATE TABLE user (
        id INT AUTO_INCREMENT NOT NULL,
        name VARCHAR(255),
        PRIMARY KEY (id)
    )
  """

  statement.executeUpdate(sqlUsers)
}
```

方法`createStatement()`返回一个实现了接口`java.sql.Statement`的对象。为执行SQL语句，使用了一个`Statement`对象。

SQL语句就是包含SQL代码的普通字符串。在这里，我们执行了SQL查询`CREATE TABLE`，它新建一个表。这个表包含两个字段：`id`和`name`，其中`id`是主键，其值必须是独一无二的，可用来快速识别记录。通过指定选项`AUTO_INCREMENT`，我们告诉H2数据库，它必须自己生成`id`值。每次创建记录时，都自动将`id`加1。字段`name`是一个简单的文本字段，最多可包含255个字符。



即便是直接使用JDBC类，使用Groovy来编写程序也可节省大量的时间。在Java中，JDBC类的方法通常可能引发`checked`异常，因此必须在`try...catch`块中调用它们，或在调用它们的方法中添加`throws`语句。Groovy不关心方法是否会引发`checked`或`unchecked`异常。

接下来需要创建一个用于存储博文的表，为此在方法`createDatabaseStructure()`末尾添加如下代码：

```
def sqlBlog = """
```

```
        CREATE TABLE blog (
            id INT AUTO_INCREMENT NOT NULL,
            title VARCHAR(255) NOT NULL,
            user INT NOT NULL,
            post CLOB,
            PRIMARY KEY (id),
            FOREIGN KEY(user) REFERENCES user(id))
    """
    statement.executeUpdate(sqlBlog)
    statement.close()
```

数据表blog包含字段user，它指向数据表user中的一条记录。user字段是一个外键。外键指向一条记录，这条记录通常位于数据库中的另一个表中。在这里，它使用用户的id字段来标识创建博文的用户。

别忘了在方法main()中调用方法createDatabaseStructure()，为此在调用createDatabaseConnection()的代码后面添加调用它的代码：

```
static void main(String[] args) {
  def app = new Main()
  def connection = app.createDatabaseConnection()
  app.createDatabaseStructure(connection)
  connection.close()
}
```

现在可以运行这个应用程序了。如果一切正常，你将看不到任何输出，因为我们还没有在代码中添加print()语句。如果看到了栈跟踪，请复查代码和SQL语句。

下面来添加一些硬编码的记录。这次我们将使用Groovy类Sql的实例来与数据库通信。请在Main类中添加如下方法：

```
def addDemoRecords(connection) {
  def sql = new Sql(connection)
  def createdUsers = sql.executeInsert("INSERT INTO user (name)
                                        VALUES (?)", ["Admin"])
  def userId = createdUsers[0][0]
  sql.execute("""
       INSERT INTO blog (title, user, post)
       VALUES (?, ?, ?)""",
       ["Test post", userId, "This is a test post"])
  sql.close()
}
```



别忘了使用组合键Ctrl + 空格（在macOS中为cmd + 空格）来补全类名Sql，并选择groovy.sql包中相应的类。

Groovy类groovy.sql.Sql的方法executeInsert返回为主键字段生成的值。由于INSERT查询可能创建多条记录，而每条记录都可能有多个主键字段，因此这个方法返回嵌套列表，其中

每个列表都包含一条记录的主键字段。在这里，我们仅为值添加了一条记录。由于数据表blog只有一个主键字段——id，因此可通过读取createdUsers[0][0]来获取生成的用户id。其中第一个索引指定了行（记录），而第二个索引指定了要读取这条记录的哪列（字段）。



与数据库连接对象一样，Sql对象也将占用宝贵而有限的数据库资源。在不再需要这种对象时，如果没有将其关闭，可能出现奇怪的问题。因此，在真实的应用程序中，总是使用try...catch块来确保即便引发了异常，这种对象也将得以妥善地关闭。

在方法main()中，在调用createDatabaseStructure()的代码后面添加调用addDemo-Records()的方法：

```
....
def app = new Main()
def connection = app.createDatabaseConnection()
app.createDatabaseStructure(connection)
app.addDemoRecords(connection)
connection.close()
...
```

运行这个应用程序，它也将在不打印任何消息的情况下退出。下面来改变这种现状：打印博文的XML表示。

12.5　使用 MarkupBuilder 生成 XML

Groovy类MarkupBuilder是一种使用Groovy动态编程功能创建的类。前一章介绍了动态地拦截方法调用，在Java和Kotlin等静态语言中，这种功能是无法如此天衣无缝地实现的。

为了演示这种功能，下面是一个使用Groovy类MarkupBuilder的示例。你无需在Eclipse IDE中输入这些代码，但可在GroovyConsole中输入并运行它们：

```
def xmlContent = new StringWriter()
def xmlWriter = new groovy.xml.MarkupBuilder(xmlContent)
xmlWriter.items {
  item(id: 1) {
    name("Item one")
  }
  item(id: 2) {
    name("Item two")
  }
}
println(xmlContent)
```

这些代码所做的工作很多，但我们先来看看结果，再详细介绍。这些代码将向控制台打印如下输出：

```
<items>
  <item id='1'>
    <name>Item one</name>
  </item>
  <item id='2'>
    <name>Item two</name>
  </item>
</items>
```

MarkupBuilder类的构造函数将一个实现了Java接口Writer的对象作为参数，生成的数据将写入到这个对象中。我们可以使用一个FileWriter实例将输出存储到一个文本文件中，但在这里，我们需要的是一个String，因此传入了一个StringWriter实例。

别忘了，在Groovy中调用函数时，括号是可选的。代码行xmlWriter.items { ... }也可写成下面这样：

```
xmlWriter.items({
  ...
})
```

在这个示例中，显然是调用了方法items()，它将一个闭包作为输入参数。MarkupBuilder类没有方法items()，但它拦截未知的方法调用和属性访问，并根据使用的方法名和参数创建XML元素。

根据 SQL 查询结果生成 XML

知道如何使用MarkupBuilder后，就可以编写根据博文记录生成XML的方法了。这个方法比较复杂，我们将分步完成其编写工作。首先来定义方法generateXML()，并添加生成XML的变量。为此，在Main类中添加如下代码：

```
def generateXML() {
  def xmlContent = new StringWriter()
  def xmlWriter = new groovy.xml.MarkupBuilder(xmlContent)
}
```

接下来定义同时获取博文数据和用户名的SQL查询。为此，在方法generateXML()的末尾添加如下代码：

```
def connection = createDatabaseConnection()
def sql = new Sql(connection)
def sqlQuery = """
  SELECT B.id, B.title, B.post, U.name AS user_name
  FROM blog B
  INNER JOIN user U ON B.user = U.id"""
sql.eachRow(sqlQuery) { record ->
}
```

在任何情况下，在一个查询中获取尽可能多的信息都是不错的主意，因为这比依次执行多个

查询的速度更快。在这里，我们使用INNER JOIN子句将数据表user和blog连接起来。这个查询返回如下列：

- ❑ blog.id；
- ❑ blog.title；
- ❑ blog.post；
- ❑ user.name。

在这个查询中，我们给user.name列指定了别名，这样可在代码中使用别名user_name来表示用户名。

我们使用Sql类的方法eachRow()来遍历返回的记录，它借鉴了函数式编程范式的做法，让我们无需手动编写循环来遍历记录，而只需指定一个将对每条返回的记录调用的闭包函数。这个闭包的代码可使用参数record来读取每天记录的值。在这个闭包中，我们将为每条记录创建一个XML条目。为此请在其中添加如下代码：

```
xmlWriter.posts {
  post(id: record.id) {
    title(record.title)
    user(record.user_name)
    def p = record.post
   post(p.getSubString(1, p.length().intValue()))
  }
}
```

这段代码你应该非常熟悉。我们创建了一个根节点为<posts>的XML。为了读取记录，只需将SQL查询中的列名（或别名，如user_name）视为属性名即可，唯一的例外是post字段。前面创建数据表时，post字段的类型被指定为CLOB，之所以这样做是因为我们不知道博文将包含多少行文本。类型为CLOB的字段的长度是不确定的，因此要读取它，必须指定每次要读取多少个字符。鉴于我们预期博文比内存量小得多，因此我们调用CLOB字段post的函数length()来指定每次要读取的字符数，从而一次性读取整个博文。



在生产级应用程序中，最好对每次读取的字符数进行限制，以避免字段消耗过多的服务器内存。为此，一种办法是小批量地读取并处理字段中的数据。

最后，我们需要关闭数据库连接，并让方法generateXML()以普通Java字符串的方式返回生成的XML。为此，在这个方法的末尾添加如下代码：

```
def generateXML() {
  def xmlContent = new StringWriter()
  def xmlWriter = new groovy.xml.MarkupBuilder(xmlContent)
  ...
  sql.close()
  return xmlContent.toString()
}
```

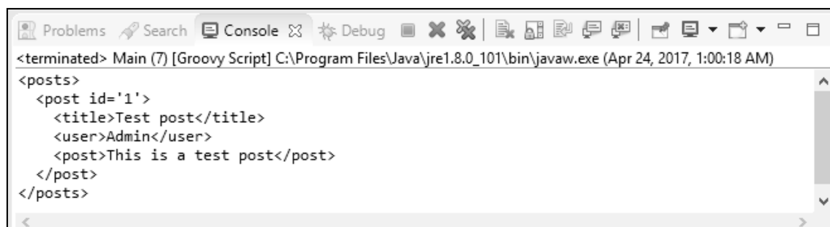
在方法main()中的代码行connection.close()后面，添加调用方法generateXML()的代码：

```
...
app.openDatabaseConnection()
app.createDatabaseStructure()
app.addDemoRecords()
connection.close()
println(app.generateXML())
...
```

你可能会问，为何要在方法中新建数据库连接，而不重用变量connection呢？这将在后面修改这个应用程序的实现，将其变成Web服务时做出解释。

如果现在运行这个应用程序，将看到如下输出：

```
<posts>
  <post id='1'>
    <title>Test post</title>
    <user>Admin</user>
    <post>This is a test post</post>
  </post>
</posts>
```



12.6　微服务平台 Vert.x

Vert.x是一个用于JVM开发的现代微型Web服务框架，最初由VMWare开发，但现在是一个Eclipse Foundation项目。Vert.x是一个货真价实的多语言框架，提供了针对Java、Groovy、Scala、Kotlin等多种JVM语言以及Nashorn（JavaScript）、JRuby和Ceylon的官方文档。

Vert.x的核心目标是高性能和可伸缩性，为此它使用了类似于Node.js的异步编程模型。简而言之，Vert.x有一个等待事件发生的主事件循环，事件发生时，它调用注册的事件处理程序对其进行处理。你在应用程序中编写的事件处理程序应尽快返回，因为这些代码执行时，Vert.x的主事件循环将无法接收并处理新事件。

如果系统中所有的事件处理程序都能快速处理事件，单个Vert.x应用程序实例就能处理数百乃至数千个请求。在Node.js和Python中，故事到这里就结束了。需要更强大的威力时，必须运行

多个独立的应用程序实例，因为这些语言无法充分利用多核CPU。这浪费了宝贵的内存和资源。Vert.x运行在JVM上，能够在不同的CPU内核中运行单个应用程序实例中的多个事件循环，从而充分发挥当今CPU的威力。为简化开发工作，一个事件处理程序通常只能在单个事件循环中使用。因此，开发人员通常根本不用考虑复杂的并发和多线程问题。



还有其他的可能性。例如，可让基于Vert.x的应用程序的多个实例组成集群，集群中的实例可以运行在同一台机器上，也可以运行在不同的服务器上。

有时候，有些操作可能需要很长时间才能完成。例如，向繁忙的数据库服务器查询并对返回的复杂数据结构进行计算时，无法保证事件处理程序将快速返回。对于这样的情形，Vert.x提供了简单的解决方案：将长时间运行的任务委托给独立的工作线程。事件循环将任务委托给工作线程，自己则继续处理其他事件。每过一段时间，它就会检查任务是否已完成，如果已完成，就根据计算得到的结果对相应的事件进行处理。由于可用的线程数是可以配置的，因此在基于Vert.x的应用程序中，能够妥善地管理可用的服务器资源（内存、线程等）。

12.6.1　在文件 ivy.xml 中添加 Vert.x 依赖

为了确定必须提供的有关Vert.x的依赖信息，我们将采用前面获悉H2数据库系统依赖信息的方法：使用Maven的在线搜索引擎（http://search.maven.org）。

为获取正确的依赖信息并将其添加到文件ivy.xml中，请执行如下操作。

- 访问前述URL并搜索vertx-core。
- 找到Groupid为io.vertx、ArtifactId为vertx-core的条目，并单击其版本号（编写本书期间为3.4.1）。
- 在产品详情（artifact details）页面中，单击Apache Ivy，并将相应的XML条目（我看到的是`<dependency org="io.vertx" name="vertx-core" rev="3.4.1" />`复制到剪贴板中：

- 在Eclipse中，打开文件ivy.xml file，并将前面复制的内容粘贴到H2数据库的<dependency>条目后面。

为让Ivy下载必要的依赖并将其添加到项目的类路径中，请执行如下步骤。

(1) 右击文件ivy.xml并选择Add Ivy Library...。
(2) 单击Finish按钮。Ivy将下载必要的依赖。

12.6.2 创建 Web 服务

首先，在文件Main.groovy的开头（package语句后面）添加必要的import语句：

```
package webservice

import java.sql.DriverManager
import groovy.sql.Sql
import io.vertx.core.AbstractVerticle
import io.vertx.core.Future
import io.vertx.core.Vertx
import io.vertx.core.http.HttpMethod
```

能够处理Vert.x事件的类被称为Verticle。通过扩展Vert.x框架提供的抽象类AbstractVerticle，可轻松地让类变成Verticle。为此，请找到下面的代码行：

```
class Main {
  ...
}
```

将其修改成下面这样：

```
class Main extends AbstractVerticle {
  ...
}
```

在抽象类AbstractVerticle中，最重要的方法是start()，它在Vert.x初始化Verticle时被调用，你应使用它来启动Vert.x内置的HTTP服务器以及注册事件处理程序。这里也将分步创建这个方法。首先来定义方法start()本身：

```
public void start(Future<Void> fut) {
}
```

传入的Future对象用于让Vert.x知道Verticle是否成功地初始化并启动了自己。在这个示例中，我们需要设置一个HTTP服务器，而设置和配置好一切可能需要一段时间。Vert.x不会等待方法start()返回，而是在Verticle初始化期间继续执行其他任务。这要求我们在启动后调用Future对象的方法complete()，而在没有启动时调用fail()。

AbstractVerticle类提供了一个名为vertx的变量，我们可使用它来与Vert.x框架通信。

下面首先来编写设置内置HTTP服务器的代码，为此在方法start()中添加如下代码：

```
Vertx
  .createHttpServer()
  .requestHandler() { request ->
}
.listen(8080) { result ->
  if (result.succeeded()) {
    fut.complete()
  } else {
    fut.fail(result.cause())
  }
}
```

抽象类AbstractVerticle提供的变量vertx指向一个实现了io.vertx.core.Vertx接口的类的实例，而这个类的每个方法都返回变量vertx指向的对象。这让你能够串接对这个对象的方法调用。要不是因为这一点，我们就必须在全部三个方法调用前面都加上vertx。

我们让HTTP服务器侦听端口8080。HTTP服务器初始化完毕后，将调用传递给方法listen()的闭包。服务器能够成功地启动时，其方法succeeded()将返回true；在这种情况下，将对传递给方法start()的Future对象调用方法complete()。这样Vert.x便知道Verticle已准备就绪了。如果HTTP服务器无法启动（例如，由于端口8080已被其他应用程序占用），将调用Future对象的fail()方法，这将导致应用程序停止执行。

方法requestHandler()指定一个HTTP请求处理程序；Vert.x主事件循环遇到HTTP请求时，将调用通过参数传递给这个方法的闭包。有一些Vert.x扩展，它们添加了复杂的路由器功能，使得能够预先注册URL，这让Vert.x能够与Express（Node.js）和Flask（Python）等流行的Web应用程序框架媲美。这里不会使用这些扩展，因为就这个简单的示例项目而言，根本不需要它们。

下面来编写HTTP请求处理程序，为此在.requestHandler(){ request -> }块中添加如下代码：

```
if (request.path() == "/blogs/" && request.method() == HttpMethod.GET){
  request
    .response()
    .putHeader("content-type", "application/xml")
    .end( generateXML().toString() )
} else {
  request
    .response()
    .setStatusCode(404)
    .end("Error 404");
}
```

request对象的所有方法都返回它，因此也可串接通过这个对象发起的方法调用。这些代码非常简单：如果请求是对URL /blogs/的GET HTTP请求，就调用返回XML数据的函数generateXML()；否则返回HTTP代码404（页面未找到）。

再来详细地说说方法generateXML()。它新建一个数据库连接、生成XML并关闭数据库连接。在大型应用程序（尤其是多线程大型应用程序）中，建议不要共享JDBC数据库连接。在这里，我们的做法是为每个请求都新建一个连接，并在处理完请求后关闭连接。就这里这样的小型示例而言，这没什么问题，但在更复杂的Web应用程序中，应使用连接池系统。新建数据库连接的开销很高，而通过使用连接池系统，可确保未用的连接被归还给连接池以便重用。在JVM平台中，可使用的连接池系统有很多。

现在余下的唯一工作是确保Verticle已启动。一种选择是直接在JVM方法static main()中完成这项任务。为此，重写方法main()，使其类似于下面这样（删除代码行println(app.generateXML())并添加突出显示的代码行）：

```
static public void main(String[] args) {
  def app = new Main()
  def connection = app.createDatabaseConnection()
  app.createDatabaseStructure(connection)
  app.addDemoRecords(connection)
  connection.close()
  Vertx vertx = Vertx.vertx()
  vertx.deployVerticle(new Main())
}
```

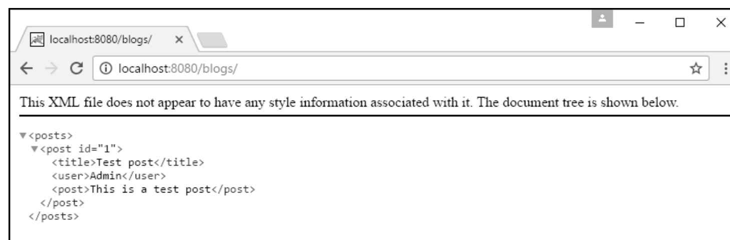
我们保留了生成数据库结构和创建示例记录的代码。在这些代码后面，我们创建一个Vertx实例，并使用方法deployVerticle来部署它。这将启动Vert.x系统，而这个系统将调用Verticle的方法start()。

现在应该可以运行这些代码了，为此可按Ctrl + F11（cmd + F11）。如果一切顺利，你将在控制台窗口中看到如下输出：

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
```

由于我们没有在项目中添加Simple Logging Facade For Java（SLF4J）依赖，Vert.x系统不知道如何写入日志。我们完成可忽略这种警告，应用程序将继续运行，而不会有其他的输出。如果出现栈跟踪，请核查代码。现在启动你喜欢的浏览器，并访问如下URL：http://localhost:8080/blogs/。

你将在浏览器中看到示例博文的XML表示：



如果你修改URL，这个Web服务将返回一个404错误页面：



要停止这个Web服务，可在选项卡Console中单击工具栏上工具提示为“Terminate”的按钮：



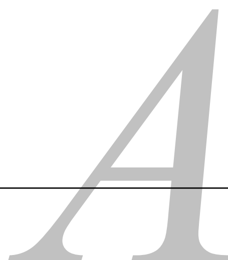
12.7　小结

本章使用Groovy和各种技术实现了一个简单的Web服务。我们首先安装了Groovy Eclipse插件以及依赖管理插件Apache IvyDE。我们在应用程序中嵌入H2 DBMS，并使用行业标准JDBC与之通信。我们创建了两个数据表，并使用一些示例记录填充它们。我们使用Groovy类`MarkupBuilde`根据数据库的内容生成了一个XML。Groovy之所以能够提供像`MarkupBuilde`这样的类，是因为它是一种动态编程语言。我们最初创建的是一个简单的控制台程序，但探索框架Vert.x后，我们将其改成了Web服务。

至此，本书要介绍的五种主要语言都介绍完了，它们是Java、Scala、Clojure、Kotlin和Groovy。但愿通过阅读本书，你找到了自己最喜欢的JVM语言。除这五种外，还有其他的JVM语言，附录A将讨论一些已经在JVM中实现的主流语言的方言，还将介绍其他一些暂露头角的语言。

附录 A

其他JVM语言



除了本书前面讨论的语言外，还有其他可在JVM中运行的语言，本附录将简要地介绍其中的一些，它们大都是流行的主流编程语言（如JavaScript、Python、Ruby和Haskell）的自定义JVM实现。本附录讨论如下语言实现：

- Oracle Nashorn（JavaScript）；
- Jython（Python）；
- JRuby（Ruby）；
- Frege（Haskell）；
- Ceylon。

A.1 Oracle Nashorn

Nashorn是Oracle推出的一种开源的服务器端JavaScript方言，从Java 8起，就包含在Java运行时环境（JRE）中，这意味着只要在主流平台（Windows、macOS、Linux和Raspberry Pi）上安装了Java 8（或更高版本），就可使用它。它取代了随Java开发包（JDK）第6版和第7版的Oracle实现提供的JVM JavaScript方言Mozilla's Rhino。

Nashorn类似于流行的使用Google V8 JavaScript引擎的服务器端JavaScript平台Node.js，它们都在服务器上运行JavaScript脚本，而不像客户端JavaScript引擎那样运行在浏览器中。Node.js和Nashorn脚本彼此不兼容，明白这一点很重要。这是因为Node.js和Nashorn都在ECMAScript语言的基础上添加了独特且不兼容的扩展。Node.js和Nashorn的一个重要不同在于，Nashorn没有实现Node.js内置的异步事件系统。



Oracle曾资助一个旨在让Nashorn与Node.js兼容的开源项目，但最后放弃了这个项目。

由于Nashorn是完全在JVM上实现的，因此与JVM库和框架兼容。它能够与Java对象交互，还能向Java库和框架传递JavaScript对象。运行JavaScript代码时，Nashorn在内部将其编译成Java字

节码，并运行在内存中动态生成的字节码；这样的情况我们在前几章使用Scala、Clojure和Kotlin的交互式shell运行包含源代码的文本文件时见到过。与这些语言一样，Nashorn也提供了一个交互式REPL shell，可用来交互地输入JavaScript代码。

有关Java 8版Nashorn的官方文档，请参阅如下URL：https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/。

Nodeclipse插件让Eclipse支持Node.js，它有一个与Nashorn兼容的版本；你可通过Eclipse Marketplace安装这个版本，让Eclipse IDE支持Nashorn。



A.1.1　在基于 JVM 的项目中嵌入 Nashorn

Nashorn的一个亮点是，在使用与Java兼容的JVM语言编写的项目中，可轻松地嵌入其引擎。在Java项目中，要支持只适用于特定客户的自定义业务逻辑验证，可让客户或经过训练的业务咨询人员输入将在运行阶段由Nashorn执行的自定义JavaScript脚本。这些脚本无需随应用程序一起发布，而是可以放在不同的目录乃至中心数据库中。



这可能带来严重的安全隐患，因为如果没有采取额外的措施，Nashorn脚本对JVM和应用程序内部的访问将不受任何限制。对外开放项目前，必须对安全实践有深入的了解。

第1章说过，JVM平台的优点之一是，可在同一个基于JVM的项目中包含使用不同JVM语言编写的代码。通过在Web应用程序的后端混合使用Java和JavaScript代码，可提供一些有趣的可能性。很多通常运行在Web浏览器JavaScript引擎上的主流JavaScript框架，不能直接在服务器端JavaScript引擎（如Nashorn）上运行。这是因为Nashorn与Node.js一样，不向JavaScript脚本提供**文档对象模型**（DOM），这不同于Web浏览器的JavaScript引擎。然而，有少量但越来越多的JavaScript库乃至经过精心设计的JavaScript代码不需要DOM，因此既能够运行在Nashorn等服务器端引擎上，又能运行在Web浏览器的客户端JavaScript上。这使得可以在服务器上运行前端JavaScript代码，并将其服务器端渲染输出提供给生成的HTML。这让搜索引擎能够看到JavaScript生成的HTML输出，而这种输出原本只有支持AJAX的Web浏览器才能看到。



DOM是Web浏览器向JavaScript引擎提供的一项功能，而不是JavaScript语言本身的功能。

JavaScript是一种并不适合用于并发编程的语言。JavaScript代码严重依赖可变的全局变量，这意味着很容易在线程中修改函数和数据，导致其他同时运行的线程出现难以发现的bug。大多数服务器端JavaScript引擎都不允许同时在多个线程中运行代码，但由于Nashorn可使用JVM线程类，因此在Nashorn中，这是可能的，不过必须仔细地规划。

A.1.2　运行 Nashorn

由于Nashorn会随Java 8或更高版本一起安装，且包含在Java开发包（JDK）和Java运行时环境（JRE）中，因此运行Nashorn易如反掌，只需执行JDK或JRE的bin目录中的一个命令即可。有关如何执行位于该目录中的命令的更详细信息，请参阅第2章。

在命令提示符或终端中，切换到JRE或JDK的bin目录，并执行如下命令：

```
jjs
```

命令`jjs`表示Java JavaScript。未指定任何命令行选项时，它将启动交互式REPL shell。这个命令还可用来运行JavaScript脚本文件，为此只需向它传递脚本的路径即可。这个命令有很多命令行选项，要查看所有的命令行选项，可使用命令行选项`-help`。

A.2　Jython（Python）

Python是一种动态语言，既易于学习又功能强大。它提供了庞大的运行时库，其生态系统也生气勃勃，这都是拜它越来越流行所赐。Python支持面向对象编程，但不要求必须这样做；另外，它还支持大量的函数式编程结构。Jython是Python的JVM实现，当前基于Python 2.7。编写本书期间，已宣布即将着手开发基于Python 3的版本。



Python 3修复了更低版中存在的大量问题，但为此付出了在很多地方都不兼容的代价。有鉴于此，多年来很多开发人员依然使用Python 2来开发项目。现在风向正在发生变化，Python开发小组可能在2020年放弃Python 2。

Jython是一种开源的Python语言实现，只能运行在JVM上。它推出的时间早于Groovy，属于第一批JVM语言。那时的Java东家Sun公司对这个项目充满期待，专门招聘了一些开发人员来开发Jython；负责Python核心的Python软件基金会也为这个项目做出了贡献。

Jython可从其主页下载：http://www.jython.org/。

要让Eclipse IDE支持Jython，PyDev插件是很不错的选择。这个插件可通过Eclipse Marketplace来安装：http://www.pydev.org/。



PyDev - Python IDE for Eclipse 5.7.0

PyDev is a plugin that enables Eclipse to be used as a Python IDE (supporting also Jython and IronPython). It uses advanced type inference techniques which... **more info**

by Brainwy Software, EPL

IDE Python Aptana Pydev Django ...

A.2.1　CPython 和 Jython 的不同之处

Python的参考实现为CPython，而名称CPython昭示着它是使用C语言编写的。CPython默认支持多线程编程，但不能在多核CPU的多个内核中运行线程。CPython使用单个CPU内核来运行所有的线程，因此它快速在线程之间切换，而不是同时在多个内核中运行线程。由于Jython基于JVM强大的线程实现，因此不存在这样的限制。另外，众所周知，与其他现代编程语言相比，CPython相对较慢，而JVM因良好而可预测的性能而受到普遍的赞誉。



相比于类似的Java代码，Jython代码的执行速度还是要慢些。这是因为Jython是一种动态语言，很多事情都是在运行阶段决定的，而在Java等静态语言中，这些决定是在编译阶段做出的。第11章对此做了更详细的解释。

Jython不能运行依赖于原生C语言库（或其他随平台而异的库）的Python代码，这意味着Jython无法使用很多流行的Python框架和库，因为它们依赖C语言库来改善性能。这种问题并非Jython独有的，其他Python实现（如PyPy）也存在这样的问题。由于Jython是一种基于JVM的语言，因此Jython代码可使用大多数基于JVM的框架和工具包。

A.2.2　运行 Jython

下载并安装Jython后，将其子目录bin添加到环境变量`Path`中。然后，就可执行如下命令来启动Jython的REPL了：

```
jython
```

要退出这个REPL，可执行函数`exit()`。

要查看所有的命令行选项，可在执行命令`jython`时加上参数`--help`。

A.3　JRuby（Ruby）

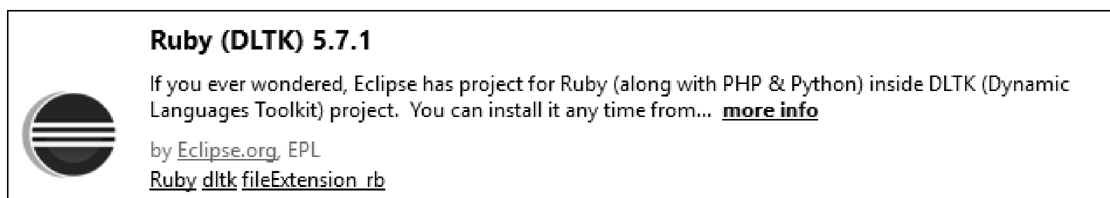
Ruby是一款流行的面向对象的动态编程语言，由于Ruby-on-Rails框架深受欢迎，很多Web应用程序都是使用它编写的。与Python一样，Ruby的参考实现也是基于解释器的，同时也是使用C语言编写的，但Ruby的面向对象程度比Python高，而且它们的语法也有天壤之别。在Python中，

几乎一切都是公有的，但Ruby像Java一样，支持访问限定符，如`private`和`protected`。

JRuby是基于JVM的Ruby实现。虽然Ruby的主要实现MRI（Maz’s Ruby Interpreter，这是根据Ruby之父Yukihiro Matsumoto命名的）发展迅速，但其他Ruby实现大都被抛弃，只有JRuby等为数不多的Ruby实现依然发展迅速。JRuby充分利用了JVM的新功能；编写本书期间，JRuby与最新的Ruby参考实现兼容。

JRuby可从http://jruby.org/网站下载。

要让Eclipse支持JRuby，可安装Eclipse动态语言工具包（Dynamic Languages Toolkit）。为此，可在Eclipse Marketplace中搜索Ruby（DLTK）：



A.3.1　Ruby on Rails 和 JRuby

Ruby on Rails是一个深受欢迎的Web开发框架，很多用于其他语言的框架都借鉴了其最初的理念。Ruby on Rails基于**模型–视图–控制器**（MVC）标准范式，并倡导**约定优先于配置**（convention over configuration）的原则，这意味着遵循Ruby on Rail的规则时，需要编写的代码更少，这在本书前面介绍过。

JRuby与Ruby的C语言扩展不兼容，因此无法使用很多常见的Ruby依赖。好消息是，Ruby on Rails框架和JRuby能够很好地协同工作，这给基于Ruby on Rails的应用程序带来了很多可能性，因为它们可以使用JVM框架乃至Java企业版（Java EE）的功能。

与Python参考实现CPython一样，Ruby的标准实现MRI不支持在不同的CPU内核中运行多个线程，但JRuby可充分利用现代CPU的所有内核。对Ruby on Rails应用程序来说，这可能是一个很大的优点，虽然程序员编写应用程序时必须考虑并发性。

A.3.2　运行 JRuby

下载并解压缩JRuby后，将其子目录bin添加到环境变量`Path`中，然后就可执行如下命令来启动JRuby的交互式控制台了：

```
jirb
```

要退出这个控制台，只需执行命令`exit`。

要查看所有的命令行选项，可执行命令`jirb --help`。

A.4 Frege（Haskell）

Frege是Haskell语言的一种方言，它无疑是第一款用于JVM的纯粹的函数式编程语言。在Frege中，函数是一等公民，可传递给其他函数；变量都是不可变的（Frege根本就没有提供赋值语句），而使用这种语言创建的方法都没有副作用。

另一个不同于Clojure的地方是，Frege是一种静态类型语言，而Clojure是一种动态类型语言。在Frege中，变量的类型是固定的且在编译阶段就必须知道。然而，在大多数情况下，Frege都能够根据代码推断出变量的类型。



有趣的是，Frege编译器将Frege源代码转换为Java代码，再调用标准的JDK编译器`javac`，将生成的Java代码转换为Java字节码。

Frege官网的地址如下（编写本书期间，访问这个网站将被重定向到其GitHub页面）：http://frege-lang.org/。

要下载编译器，请访问其GitHub页面（http://github.com/Frege/frege/releases）。

Frege是一种相对较新的语言，其工具集还不太完备。虽然有用于Eclipse IDE的插件，但本书出版时，该插件与最新的Eclipse版本不兼容。Frege的wiki页面（可在其GitHub页面中找到）指出，使用Frege进行开发时，JetBrains的IntelliJ IDEA IDE是不错的选择，这种说法应该对IntelliJ IDEA的免费社区版和收费版都适用。

A.4.1 在 Frege 中调用 Java 代码

JVM本身并没有遵守纯粹的函数式编程规则：Java类库中很多的内置类都是可变的，流行的JVM框架和库提供的大多数类亦如此。有鉴于此，Frege设计者必须找出规避方案，让Frege代码能够调用不纯粹的JVM方法。

由于没有可靠的方法来自动检测方法是否具有副作用（它可能直接修改实例变量，也可能调用对实例变量进行修改的方法），程序员必须向Frege指出方法是否是**纯粹的**（没有副作用）。如果JVM方法被宣称为纯粹的，就可像调用其他Frege函数一样调用它；而对于被声明为包含状态的方法，就使用内置的monad来调用它。monad将返回一个不可变值，这个值是由方法计算得到的。可变数据放在monad里面，这样程序就无法直接访问它了。

A.4.2 运行 Frege

Frege是以单个可执行的JAR文件的方式发行的。当前，标准发行版中没有自带REPL，但可从http://github.com/Fre ge/frege-repl/releases单独下载它。

请下载最新版本的REPL，并将ZIP文件解压缩。可将其解压缩到包含Frege编译器的JAR文件所在的目录。然后，切换到其中的子目录bin，并执行如下命令来启动REPL：

```
frege-repl
```

要退出这个shell，可执行命令`quit`。

另外，对于非常简单的程序，也可使用在线REPL：http://try.frege-lang.org/。

A.5 Ceylon

Ceylon也是一种面向对象的静态类型语言，由对Java及其生态系统有深入了解的Red Hat公司开发。与本书介绍的其他一些语言一样，除JVM外，Ceylon也可将代码编译为其他目标；例如，可将Ceylon代码编译为客户端JavaScript代码（以便在Web浏览器中运行）或服务器端JavaScript代码（使用Node.js）。

Ceylon提供的功能与Kotlin很像；它们都是静态类型语言，都在面向对象的同时提供了函数式编程功能，而且都提供了一个确保null安全的类型系统。Ceylon特有的一种功能是支持模块化应用程序。Java 9引入了新的模块系统Jigsaw，而Ceylon支持模块系统JBoss。



JBoss是Red Hat的一家子公司，因此它选择这个模块系统就没什么可奇怪的了。在Red Hat推出的Java EE应用程序服务器WildFly和相关的产品中，大量地使用了JBoss模块。

Ceylon网站的地址为http://ceylon-lang.org。

要让Eclipse支持Ceylon，可通过Eclipse Marketplace安装Ceylon IDE插件：

Ceylon IDE 1.3.1

Ceylon IDE provides eclipse support for Ceylon (http://ceylon-lang.org). This is a full-featured development environment for Ceylon, including interactive error... **more info**

by Red Hat, Other Open Source

ceylon fileExtension_ceylon

A.5.1 Ceylon 的模块系统

虽然在第9版之前，Java和JVM一直没有内置的模板系统，但它们支持JAR文件。第2章说过，

单个JAR文件可包含多个类文件，模块系统又能添加什么功能呢？JAR文件缺少的一项重要功能是，定义版本信息和依赖。

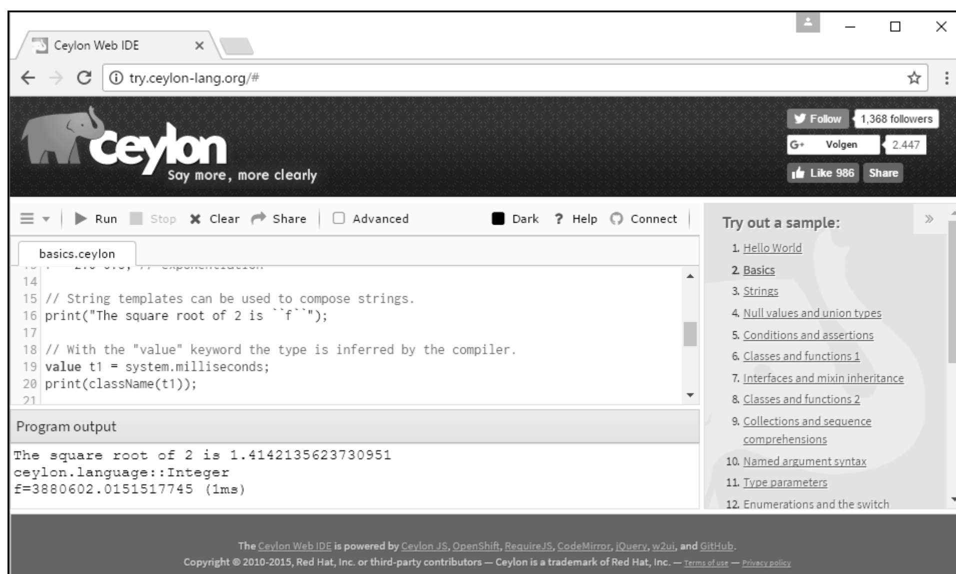
为改变这种状况，构建工具Apache Maven定义了一个XML对象模型，让库（它们通常是以JAR文件的方式发布的）能够指定其依赖。如果库在Maven文件build.xml中指定了依赖，Maven（或其他带依赖管理器的构建工具，如Gradle）将下载这些依赖（包括这些依赖的依赖），并将它们都放到项目的类路径中。

然而，成熟的模块系统还有其他功能。在良好的模块系统中，应该能够指定要向模块的使用者暴露哪些代码。对于仅供内部使用的代码，应禁止从外部使用它们。虽然Java支持将类的成员声明为私有的，但无法隐藏公有类（应该让其他包能够使用的类），即便这些类是仅供当前模块内部使用的。

Ceylon内置的模块系统支持存储版本信息、指定依赖，并为类在模块外部的可见性方面提供了大得多的控制权。另外，Ceylon开发小组还维护着一个免费的在线仓库——Ceylon Herd，你可通过它下载和分享Ceylon模块。Ceylon编译器和工具全面集成了其模块系统；实际上，Ceylon JVM编译器不会将代码编译为不同的类文件，而总是为项目创建一个模块。

A.5.2　运行 Ceylon

当前，Ceylon没有自带REPL，但你可使用Ceylon网站（http://try.ceylon-lang.org）提供的在线REPL。



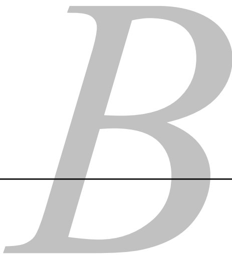
A.6　小结

在这个附录中，我们讨论了一系列其他的JVM语言：Oracle Nashorn（JavaScript）、Jython（Python）、JRuby（Ruby）、Frege（Haskell）和Ceylon（一种独特的语言）。对于每种语言，我们都讨论了其母语言，并介绍了一个特别有趣或独特的功能或库；我们还指出了要在Eclipse IDE中支持它需要安装的插件（如果有的话），并介绍了如何使用交互式REPL shell（如果有的话）来运行它。

本书到这里就结束了。JVM是一个功能强大而稳定的虚拟机，支撑着很多深受欢迎的在线服务和拥有全球数百万用户的网站，并有望充分利用新出现的技术和趋势。但愿通过阅读本书，你对JVM本身以及一些重要的JVM语言有更深入的认识。

附录 B

小测验答案



第 3 章

问　题	答　案
1	c
2	b
3	a
4	b
5	d

第 5 章

问　题	答　案
1	b
2	a
3	c
4	c
5	b

第 7 章

问　题	答　案
1	b
2	c
3	a
4	b
5	a

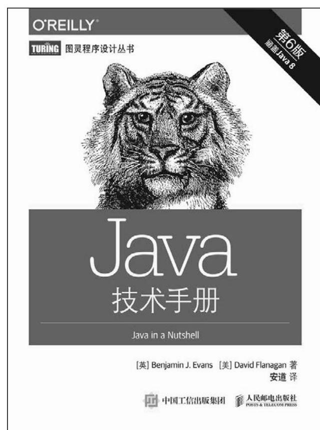
第 9 章

问　题	答　案
1	b
2	b
3	d
4	b
5	c

第 11 章

问　题	答　案
1	d
2	a
3	b
4	c
5	b

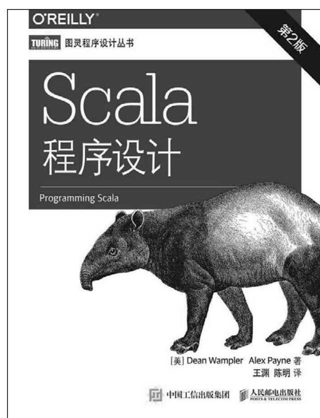
技术改变世界 · 阅读塑造人生



Java 技术手册（第 6 版）

- O'Reilly重头Java图书
- 快速准确地介绍Java编程语言和Java平台
- 讲解核心概念和API，展示如何在Java环境中解决实际的编程任务

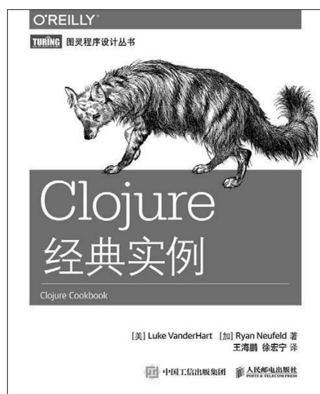
作者： Benjamin J Evans David Flanagan
译者： 安道



Scala 程序设计（第 2 版）

- 全面展示Scala语言生态环境下，高效编写代码的方法与技巧
- 涵盖Scala最新语言特性，新添了模式匹配、推导式以及高级函数式编程
- Typesafe顾问Dean Wampler、Twitter平台负责人Alex Payne作品

作者： Dean Wampler Alex Payne
译者： 王渊 陈明

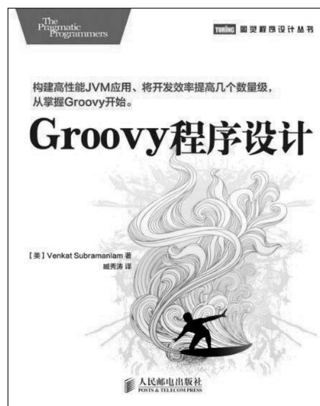


Clojure 经典实例

- 全面经典的Clojure开发指南
- 涵盖150多个实例，源于全球60多名顶级Clojure开发者
- 解决方案全面广泛：从构建动态网站和应用数据库到网络通信、云计算、高级测试策略等

作者： Luke VanderHart Ryan Neufeld
译者： 王海鹏 徐宏宁

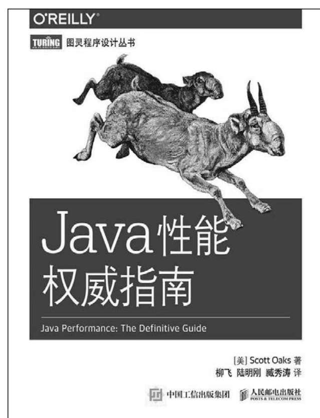
技术改变世界 · 阅读塑造人生



Groovy 程序设计

- **全面实用的Groovy程序设计指南**
- **既涵盖Groovy编程基础，又涉及该语言的最新高级特性**
- **具备Java基础的程序员掌握Groovy的首选图书**

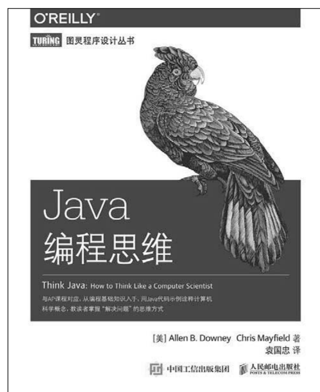
作者： Venkat Subramaniam
译者： 臧秀涛



Java 性能权威指南

- **深入理解Java平台性能，让你的程序如虎添翼！**

作者： Scott Oaks
译者： 柳飞 陆明刚 臧秀涛



Java 编程思维

- **与AP课程对应，从编程基础知识入手，用Java代码示例诠释计算机科学概念，教读者掌握“解决问题”的思维方式**

作者： Allen B. Downey , Chris Mayfield
译者： 袁国忠



微信连接



回复“Java”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区

iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈



Java虚拟机（JVM）是开发和部署软件的成熟的现代平台，最初只有Java一门语言运行于其中。随着Java的日益复杂以及JVM性能的增强，出现了新一代可在JVM中运行的编程语言。

本书首先概述JVM及其特性，并介绍至关重要的JVM概念。接下来介绍Java、Scala、Clojure、Kotlin和Groovy这五种基于JVM的语言，分别探讨它们的特性和用例，并通过使用它们编写示例项目来展现各自的优缺点，以便帮读者找出能满足特定需求的语言。

- 了解JVM基本概念
- 熟悉流行的JVM语言及Java类库
- 掌握命令式、面向对象的和函数式等编程范式
- 使用Eclipse IDE、Gradle、Maven等常见JVM工具
- 探索SparkJava、Vert.x、Akka、JavaFX等框架
- 了解主流编程语言（JavaScript、Python、Ruby和Haskell）的JVM实现

[PACKT]
PUBLISHING

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/程序设计/Java

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-47779-8
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks